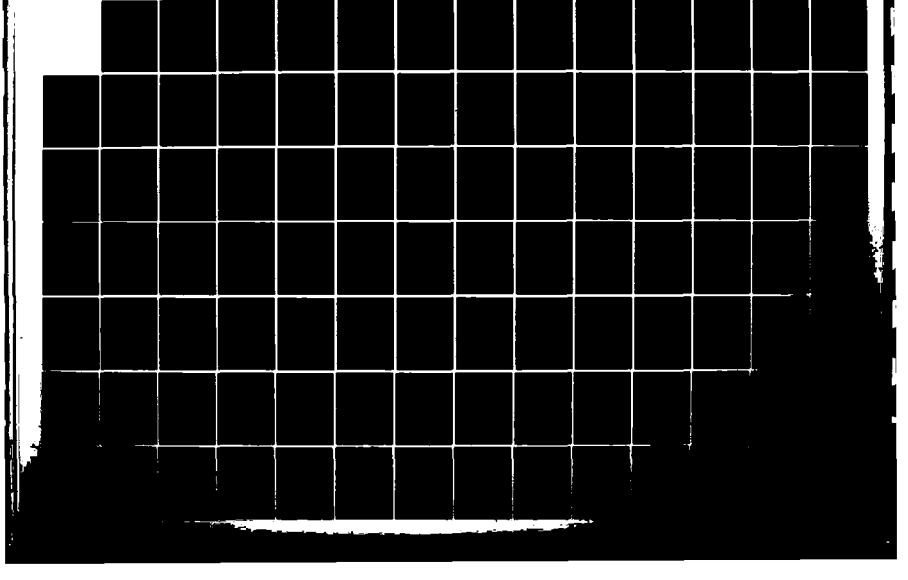
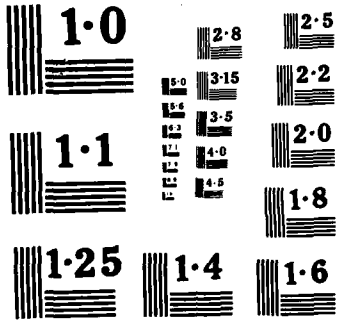
AD-A156 314 DESIGN FOR MISP: A MULTIPLE INSTRUCTION STREAM SHARED PIPELINE PROCESSOR(U) ILLINOIS UNIV AT URBANA COMPUTER SYSTEMS GROUP L M PEDERSEN DEC 84 CSG-37 1/3

UNCLASSIFIED N00039-80-C-0556 F/G 9/2 NL





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

CSG-37 DECEMBER 1984

(2)

AD-A156 314

DESIGN FOR MISP: A MULTIPLE INSTRUCTION STREAM SHARED PIPELINE PROCESSOR

LYNNE MARIE PEDERSEN

DTIC FILE COPY

DTIC ELECTE JUL 0 8 1985 S D E

for ... distribution ...

85 06 25 055

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS: None
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for distribution to all DTIC users.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): CSG-37	5. MONITORING ORGANIZATION REPORT NUMBER(S): N/A
6a. NAME OF PERFORMING ORGANIZATION: Coordinated Science Lab. University of Illinois 61801 / 6b. OFFICE SYMBOL (If applicable): N/A	7a. NAME OF MONITORING ORGANIZATION: Office of Naval Research
6c. ADDRESS (City, State and ZIP Code): 1101 W. Springfield Ave. Urbana, Illinois 61801	7b. ADDRESS (City, State and ZIP Code): 2511 Jefferson Davis Highway Arlington, Virginia 22202
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Naval Electronics Sys. Comm. / 8b. OFFICE SYMBOL (If applicable): N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00039-80-C-0556
8c. ADDRESS (City, State and ZIP Code): 2511 Jefferson Davis Highway Arlington, Virginia 22202	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
N/A	N/A	N/A	N/A

11. TITLE (Include Security Classification): DESIGN FOR MISP: A MULTIPLE INSTRUCTION STREAM SHARED PIPELINE PROCESSOR

12. PERSONAL AUTHOR(S): LYNNE MARIE PEDERSEN

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM ___ TO ___	December 1984	191

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Computer Architecture, Multiprocessing, Pipelined, Multi-stream

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Advancements in VLSI technology have provided new horizons for growth in computer architecture. Due to the increased circuit density of integrated circuits, single-chip microprocessors can now include multiprocessor architectures that can utilize the increased capacity of a chip more effectively. The Multiple Stream Shared Pipeline Processor is an eight-segment pipelined processor and is designed to allow eight essentially independent instruction streams to execute concurrently. This allows the hardware resources of the pipeline to be shared among the eight instruction streams.

The basic hardware design for the MISP processor is presented. Methods for accomplishing process control are discussed and a description of the operating system is given. A mechanism for trap and interrupt handling is presented and the control structure is described. An appropriate system configuration is given for the MISP processor. The feasibility of a VLSI implementation of the MISP processor is evaluated.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☒	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
		None

MISP has several advantages over a conventional microprocessor implementation. MISP has a higher pin utilization over a single processor, since it has one address bus and one bidirectional data bus which is shared by all of the processes. It also maximizes hardware resource utilization, since much of the hardware in the segments is shared and only the process state needs to be replicated for each process. Thus, for a modest cost increase, the MISP processor architecture achieves nearly an eight times speed-up over a conventional single stream microprocessor architecture. MISP represents a cost-effective alternative to conventional multiprocessor architectures.

DESIGN FOR MISP:

A MULTIPLE INSTRUCTION STREAM

SHARED PIPELINE PROCESSOR

BY

LYNNE MARIE PEDERSEN

B.S., University of Washington, 1981

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

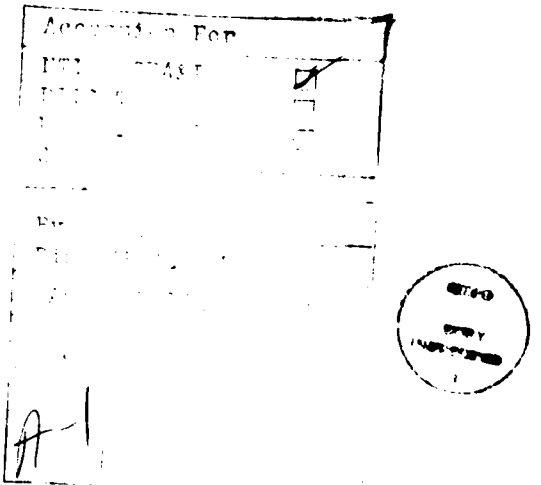
Urbana, Illinois

ABSTRACT

Advancements in VLSI technology have provided new horizons for growth in computer architecture. Due to the increased circuit density of integrated circuits, single-chip microprocessors can now include multiprocessor architectures that can utilize ıe increased capacity of a chip more effectively. The Multiple Stream Shared Pipeline Processor is an eight-segment pipelined processor, and is designed to allow eight essentially independent instruction streams to execute concurrently. This allows the hardware resources of the pipeline to be shared among the eight instruction streams.

The basic hardware design for the MISP processor is presented. Methods for accomplishing process control are discussed and a description of the operating system is given. A mechanism for trap and interrupt handling is presented, and the control structure is described. An appropriate system configuration is given for the MISP processor. The feasibility of a VLSI implementation of the MISP processor is evaluated.

MISP has several advantages over a conventional microprocessor implementation. MISP has a higher pin utilization over a single processor, since it has one address bus and one bidirectional data bus which is shared by all of the processes. It also maximizes hardware resource utilization, since much of the hardware in the segments is shared, and only the process state needs to be replicated for each process. Thus, for a modest cost increase, the MISP processor architecture achieves nearly an eight times speed-up over a conventional single stream microprocessor architecture. MISP represents a cost-effective alternative to conventional multiprocessor architectures.



ACKNOWLEDGMENT

I would like to express my sincere appreciation to my advisor, Dr. Edward S. Davidson, for his invaluable contribution to this research. Dr. Davidson's patient guidance, technical insight, and unselfish donation of time and energy were indispensable to this work. I would also like to thank the other professors in the Computer Systems Group, Jacob A. Abraham, Janek H. Patel, and Ravi K. Iyer, as well as my co-workers and office mates, for their helpful suggestions, lively discussions, and for providing an intellectually stimulating environment. I also appreciated the clerical help of Marty North and Jacqueline Ziemer, as well as their encouragement and friendship.

I am grateful for the love and support from my parents, family, and friends, which encouraged me throughout the progress of this research. Finally, I would like to thank the Lord, for the strength and the grace to complete this work.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. OVERVIEW OF THE MISP PROCESSOR

1.1. Background

The VLSI revolution has produced new horizons for computer architecture. With the improvement of VLSI technology and the corresponding increased circuit density, the complexity of the hardware that can fit on a single silicon chip has increased considerably. Single-chip microprocessor architectures can now include multiprocessor architectures that can utilize the increased circuit capacity of a chip more effectively.

We have developed the Multiple Instruction Stream Shared Pipeline (MISP) processor, a multiprocessor architecture intended to fit on a single chip and provide high resource utilization. The MISP architecture concurrently executes eight instruction streams with very few data dependencies between instruction streams. Critical sections of code and memory mailboxes are supported for handling these dependencies, but each stream can run nearly independently at full speed. This architecture maximizes resource sharing to utilize the silicon area efficiently and thereby attain increased cost-effectiveness. MISP also minimizes the pin count and associated chip area by sharing pins among all instruction streams. Pin sharing also decreases the complexity of the external interconnection network with respect to a conventional multiprocessor. The Multiple Instruction Stream Shared Pipeline (MISP) processor is a multiprocessor architecture that should be seriously considered for future VLSI processors.

The MISP architecture originated from previous research. Shar performed the original cost-effectiveness analysis of shared pipelined multiprocessor architectures, [SHA74]. Later work was performed at the University of Illinois. Kaminsky proposed a methodology for designing multiple instruction stream shared pipeline microprocessors, [KAM77] and [KAM79]. Emer examined shared resources for pipelined systems and analyzed their performance. He also developed a scheme for control store organization for shared pipeline processors, [EME78] and [EME79]. A prototype system based on eight

Motorola 6800 processors was implemented to emulate a shared pipeline processor system, [DAV80].

The MISP architecture was developed by Archer [ARC82], based on a DEC PDP-11 instruction set, [DEC81]. He proposed a basic two-pipeline organization (for execution and process control, respectively), and defined the major functional units for the system. In this work, we further develop the architecture by completing the control section of the processor and integrating the process control function into the main pipeline. We have augmented the PDP-11 instruction set to support the MISP multiprocessing environment and have developed guidelines for the MISP operating system.

1.2. Basic Architecture

The major functional blocks of this processor have been separated into eight pipeline segments, as seen in Figure 1.1. There are eight instruction streams executing concurrently in these segments, one stream in each segment. Each stream proceeds sequentially through each segment, looping back after Segment Eight to Segment One in a round robin fashion. This flow allows the major hardware resources to be specialized to their particular tasks and to be shared among the streams.

The register set of the MISP processor supports the MISP instruction set, which is similar to the PDP-11 instruction set. The instruction set is described in Section 1.4. The MISP register set for one stream consists of eight sixteen-bit general registers--R0-R7. R6 is generally used as the Stack Pointer, and R7 is used as the Program Counter. Each stream also uses one sixteen-bit Process Status Register and one sixteen-bit Instruction Register (IR). Each of these registers must be replicated for each stream. We have done this by implementing a dynamic shift register which contains registers R0-R6 for each stream. These registers are then shifted down through the register file, in parallel with their associated stream, as the stream travels through the pipeline. The register file has a dual read port in Segment Two and a single write port in Segment Seven. Each stream can read or write its own registers when it reaches the appropriate segment. This register organization minimizes hardware since dynamic logic can be used and register ports are shared and need not be replicated for each stream.

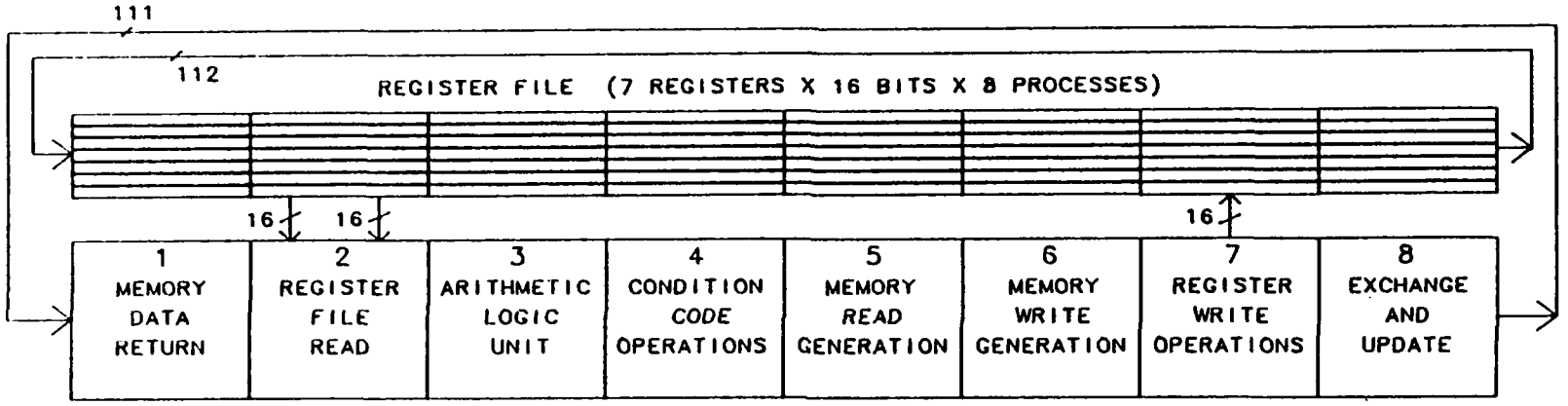


Figure 1.1 MISP Processor Pipeline Organization

There are some additional user-transparent registers. The contents of these registers travel through the pipeline with the stream. The Program Counter (PC), the Instruction Register (IR) and the Process Status Word (PS) pass through the pipeline with these registers. Other registers that pass through the pipeline include three temporary memory data registers (MD1-MD3), the Dynamic Status Register (STAT), the Stream Identification number (SID) and the Cycle Counter (CT). The implementation and use of these registers are fully described in Chapter 2.

There are one sixteen-bit bidirectional data port for memory and one address port to memory. There is a Memory Read Buffer (MRB) in Segment Five and a Memory Write Buffer (MWB) in Segment Six. These buffers store memory read requests or memory write requests, respectively. The address bus is shared by both the memory read requests and the memory write requests. There is a Memory Data Buffer (MBUF) in Segment One, where the data that has been read from memory is returned. The data bus is shared by both the memory write request and the return of the read data from memory. These buses are described in more detail in the next section. The use of these buffers to store requests and to share the data and address buses maximizes the use of the pins, and optimizes the use of the memory bandwidth.

The basic process flow is as follows. An instruction cycle (or cycle) refers to one pass through the entire pipeline, i.e., eight clock cycles. In Segment One, the process can pick up a new instruction or read data that was requested in the previous cycle. In the second segment, it can read one or two operands from the register file or it can read the address of an operand. In addition, in the first cycle of each instruction, the Cycle Counter (CT) is set to the number of cycles in that instruction. In Segment Three, arithmetic and logic operations can be performed on the temporary registers traveling in the pipeline with the process. Data or address calculations can be performed. In Segment Four, the condition codes can be updated after ALU operations. In Segment Five, a memory read request for a new instruction or for data can be issued. In Segment Six, memory write requests are issued to store results or perform a stack push. In Segment Seven, results can be written into the register file. In Segment

Eight, the Cycle Counter is decremented. In addition, values in certain temporary registers can be swapped.

With these eight segments, it is possible to obtain a speed-up of eight over a conventional serial processor when no wait cycles are needed. A wait cycle is required if the memory can not return the data from a read request by the time the stream arrives at Segment One to pick up its data. If this occurs, the stream passes through the pipeline once again, executing NOP's in each segment until it arrives at Segment One again. If read-wait passes are common, adding dummy segments to the pipeline can increase the performance per unit cost of the processor, as shown in [ARC82]. The performance per unit cost of the processor is optimized when just enough dummy segments are added to allow the instruction stream to initiate a memory request in Segment Five and arrive in Segment One just as its requested data has returned from memory. This condition is shown by the equation below, where m is the memory response time, s is the total number of segments in the pipe, and c is the processor clock period:

$$m \leqslant (s - 5) * c$$

1.3. I/O Pins

The I/O pins for the processor are shown in Figure 1.2 and listed in Table 1.1. The address pins are shared by the memory read requests and the memory write requests. The Address Stream I. D. is used to identify either of these requests as it is sent to memory. The data pins are bidirectional and are shared by the memory write requests and the data return. The Data Stream I. D. is used to tag the data returning from memory. With this scheme, the memory read and write requests share the same pins. This scheme maximizes the pin utilization. In addition, the read requests can be overlapped with the return of previously requested read data from memory. ERR is sent when the data is returned from

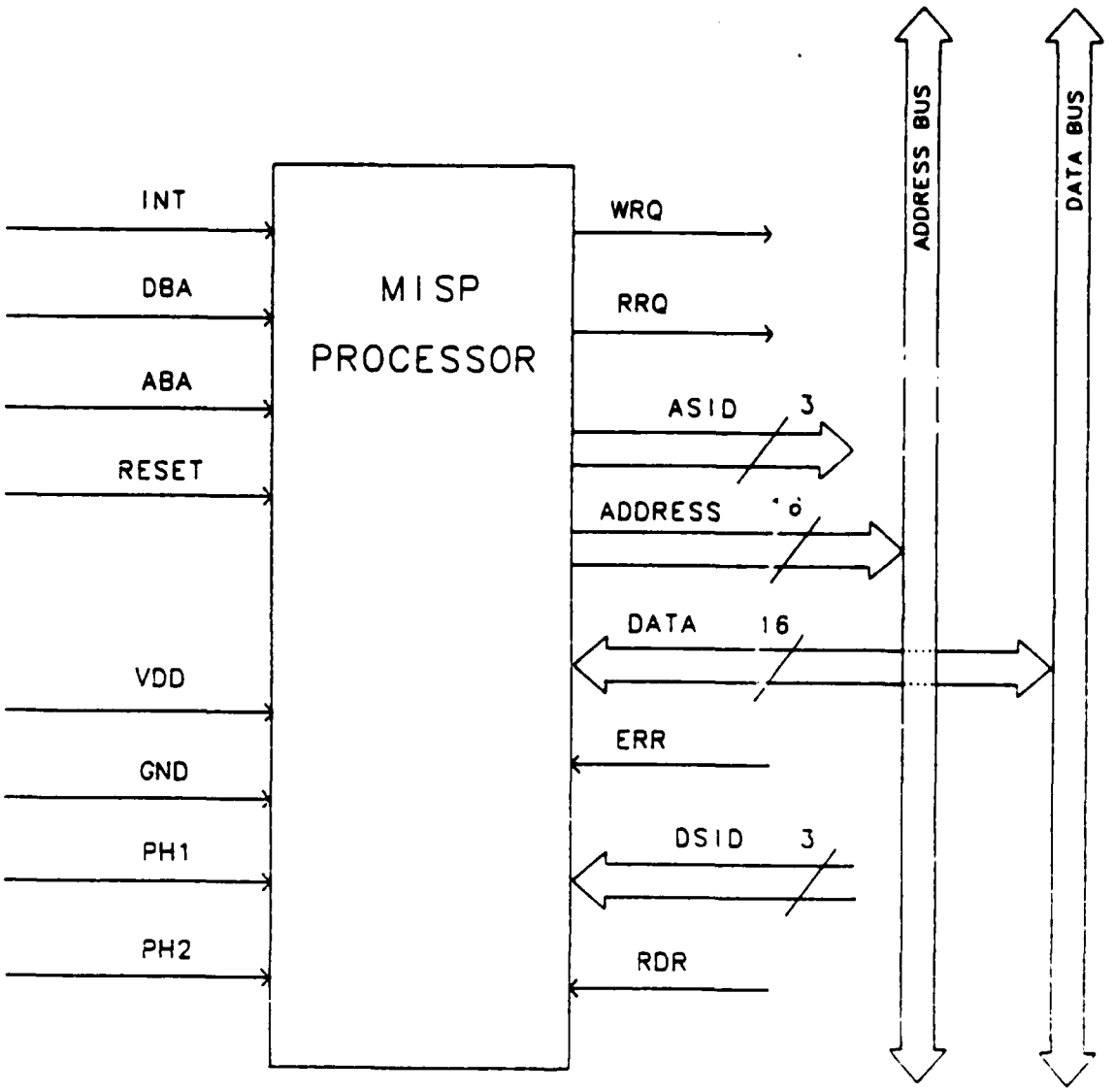


Figure 1.2 I/O Pins for MISP Processor

Table 1.1 I/O Pins

No. of Pins	Name	Description
16	D0-D15	Data pins
16	A0-A15	Address pins
3	DSID	Data Stream I. D.
3	ASID	Address Stream I. D.
1	ERR	Memory error
1	RRQ	Read Request
1	WRQ	Write Request
1	ABA	Address Bus Available
1	DBA	Data Bus Available
1	RDR	Read Data Return
1	RESET	Reset pin
1	INT	Interrupt pin
1	VDD	Power
1	GND	Ground
1	PH1	Phase 1 clock
1	PH2	Phase 2 clock
50		Total no. of pins

memory, and indicates whether a memory error has occurred. RRQ and WRQ are signals sent by the processor indicating that the processor is requesting the use of the buses to make a memory read or a memory write request, respectively. ABA and DBA are control signals sent from a bus controller or generated from a daisy chain priority system which indicate that the address bus or data bus is available, respectively. RDR is a signal coming from memory indicating that data is being returned from memory. RESET is a pin that can be pulsed to reset the processor. INT is a pin that can be pulsed to interrupt the processor. Power and ground pins and two clock signals are included. The implementation and use of these pins are described in Chapter 2. These signals provide for a flexible processor that can interact in a variety of reasonable system configurations.

1.4. Instruction Set

We modified the instruction set originally proposed for MISP by Archer [ARC82] to enhance its capabilities for supporting the MISP multiprocessing environment, as well as its general program execution capability. The original proposal was a modified version of the PDP-11 instruction set. The PDP-11 instruction set was proposed because it is an orthogonal instruction set, which should provide regularity. It is also a familiar instruction set with a large established software base.

Instead of the eight modes of addressing used in the PDP-11, Archer proposed only three. We added one additional addressing mode to the three already proposed so that all of the other original PDP-11 modes of addressing could be implemented in a straightforward fashion with simple MISP routines. These four modes of addressing, register direct, register deferred (indirect), autoincrement mode, and indexed mode, are shown in Table 1.2. In register direct mode, the selected register contains the operand. In register deferred mode, the selected register contains the address of the operand. In autoincrement mode, the selected register contains the address of the operand, and the register value is automatically incremented by two after the operand is fetched. Autoincrement mode was added due to

Table 1.2 Modes of Addressing

Mode Number	Addressing Mode
0	register direct, Ri
1	register deferred (indirect), (Ri)
2	autoincrement, (Ri)+
3	indexed, n(Ri)

its general utility for scanning structures and because it can be used for Program Counter Immediate mode when the Program Counter (PC) is used as the selected register. When this mode is used, the immediate operand is located in line after the current instruction, so that the Program Counter points to the operand. The Program Counter is then automatically incremented to point to the next instruction. In indexed mode, an offset, n, is added to the value of a register to produce the effective address of an operand. This offset is located directly after the instruction in the instruction stream. This mode is often used when addressing a specific item in a table relative to a starting address for that table. Each of these addressing modes and the other addressing modes of the PDP-11 are fully described in the PDP-11 Processor Handbook [DEC81].

An implementation of the other modes of addressing in the PDP-11 instruction set with these four basic modes is shown in Figure 1.3, using the Complement base instruction as an example. As can be seen, the implementation of autoincrement deferred mode requires two instructions and one additional register. The implementation of autodecrement mode requires two instructions. The

Autoincrement deferred mode

```
PDP-11
COM     @(R2)+      /*R2 is used as the address of the address of the operand, and
                    /*then R2 is incremented by 2. The operand is complemented.

MISP
MOV     (R2)+,R0    /*Move the address of the operand to R0.
                    /*Autoincrement R2 by 2.
COM     (R0)        /*Complement the operand indirectly through R0.
```

Autodecrement mode

```
PDP-11
COM     -(R3)       /*The register contains an address that is automatically decremented
                    /*by 2. The operand is then complemented using this address.

MISP
SUB     2,R3        /*Decrements the register by 2.
COM     (R3)        /*Complements the operand indirectly through R3.
```

Autodecrement deferred mode

```
PDP-11
COM     @-(R4)      /*Contents of R4 are decremented by 2 and then used as the address
                    /*of the address of the operand. The operand is then complemented.

MISP
SUB     2,R4        /*Contents of R4 are decremented by 2.
MOV     (R4),R0     /*Move the address of the operand to R0.
COM     (R0)        /*Complement the operand indirectly through R0.
```

Index deferred mode

```
PDP-11
COM     @1000(R5)   /*1000 and the contents of R5 are summed to produce the
                    /*address of the address of the operand. The operand is complemented.

MISP
MOV     1000(R5),R0 /*1000 and the contents of R5 are summed to get the address of
                    /*the address of the operand. R0 gets the address of the operand.
COM     (R0)        /*Complement the operand indirectly through R0.
```

Figure 1.3 Implementation of PDP-11 Addressing Modes

implementation of autodecrement deferred mode requires three instructions and one additional register. The implementation of index deferred mode requires two instructions and one additional register. A translator can convert PDP-11 programs to MISP programs in a straightforward fashion.

In addition to adding one mode of addressing, we also added some instructions which facilitate the multiprocessor environment. These instructions add some control features to allow an operating system to function properly. The new instructions are primarily in the Process Control class and the System Trap class. These instructions are listed in Table 1.3 and their use is explained in Chapter 3. The complete instruction set is described in Appendix A, with the modified instructions described in greater detail. Complete descriptions of the other instructions can be found in the PDP-11 Processor Handbook [DEC81].

1.5. Process Execution

This section gives a brief overview of how processes are executed on the processor. There are eight instruction streams executing concurrently on the processor. Many other processes could be waiting to be run in the pipeline. These processes wait in the Process Ready Queue (PRQ) in memory. An instruction stream will generally execute in the pipeline until it is finished. When it is finished, it traps to the operating system. It can then be replaced by another process found in the Process Ready Queue in memory. The operating system selects one of the ready processes and swaps it into the terminated stream's position for execution. A description of the operating system and process swapping is given in Chapter 3.

Each instruction requires one or more passes through the pipeline (cycles). For each instruction stream, the instructions from the stream are executed serially. However, an external interrupt can cause a low priority process to be interrupted between instructions. When an interrupt occurs, the status of the process is saved, and the interrupt is serviced by an interrupt service routine. Traps also interrupt

Table 1.3 Process Control and System Trap Instructions

Process Control Instructions

Instruction	*Description*
RESET	Resets processor, halts processes. (Kernel Mode only)
SPL	Set priority lower.
HLTP	Halts another process. (Kernel Mode only)
CPSW	Change Process Status Word to new word. (Kernel Mode only)
RPSW	Read Process Status Word
TSET	Test and set semaphore register (Kernel Mode only)
CTST	Clear semaphore register (Kernel Mode only)

System Trap Instructions

INTR	Interrupt Instruction
Halted Process Instruction	Trap for a process that has been halted
Trace Trap Instruction	Trap for tracing instruction streams
Illegal Instruction Trap	Illegal or reserved instructions
Memory Error Trap	Trap due to memory errors
START Trap	Trap for execution of a startup after a system reset

the serial execution of the instruction stream. It is possible to trace an instruction when the Trace is enabled (the T bit in the PS is set), which causes a trap to the operating system after each instruction in the stream. Other traps occur as a result of error conditions, such as the execution of an illegal instruction or the occurrence of a memory error. These error conditions cause the execution of the process to be terminated, and another process is swapped into its place. It is also possible for one process to halt another process (a new process will then start in its place). The execution of traps and interrupts is described in detail in Chapter 3.

Chapter 2 describes the hardware design for this system. We describe the functions of the registers and introduce the register transfer language used to describe the instruction set. The hardware for each of the segments is then described in detail. Chapter 3 presents an overview of the operating system functions. Methods for accomplishing process control and process swapping for the system are discussed including the trap and interrupt mechanisms. An appropriate system configuration for the MISP processor is also given in Chapter 3. An estimate of the total circuit count for possible VLSI implementation and conclusions are presented in Chapter 4.

2. DESIGN DESCRIPTION

2.1. Register Organization

A separate set of registers is associated with each stream. These registers contain the state of the process that is running in that stream. Table 2.1 contains a list of the registers in each set.

The six general purpose registers and R6 (the Stack Pointer) for all streams are located in a dynamic register file that is shifted in parallel with the pipeline. This dynamic register file is connected to the pipeline with a dual read port at Segment Two and one single write port at Segment Seven.

Table 2.1 Register Organization

SIX GENERAL PURPOSE REGISTERS	R0-R5	16 bits each
STACK POINTER	R6 (SP)	16 bits
PROGRAM COUNTER	R7 (PC)	16 bits
INSTRUCTION REGISTER	IR	16 bits
THREE TEMPORARY STORAGE REGISTERS	MD1-MD3	16 bits each
PROCESS STATUS REGISTER	PS	16 bits
DYNAMIC STATUS REGISTER	STAT	9 bits
STREAM IDENTIFICATION NUMBER	SID	3 bits
CYCLE COUNTER	CT	3 bits

The rest of the registers travel through the pipeline with each stream. These registers contain data and information about the stream that are used in many of the stages of the pipeline. The Program Counter (PC), which is also addressed as R7, contains the address of the next instruction for the stream. It is located with the registers in the pipeline for special use as the PC, although it can also be accessed in an instruction as R7. The Instruction Register (IR) contains the instruction that is currently being executed. Figure 2.1 shows an example of the Instruction Register for the Double Operand (DOP) instructions. The exact format for the Instruction Register varies for different instruction classes. The formats for the Instruction Register for all the instruction classes are included in Appendix C. Three temporary storage registers (MD1-MD3) contain data that is used to store temporary data until the data can be used or stored in a register or memory.

The remaining registers carried through the pipeline are used for process status, identification and control. Figure 2.2 shows the format of the Process Status Word (PS). The Process Status Word contains information that needs to be stored when the process is swapped out of the pipeline. It consists of an eight-bit Process Identification number (PID), the Priority flag (P), two unused bits (**), the Trace bit (T), and four condition code flags. The eight-bit PID is used to identify the process which is running in that particular stream. In addition to the running processes, there could be many other processes in memory queues. These processes can be swapped in and out of the pipeline. The PID allows 256 processes to be uniquely identified. The PS also contains a Priority bit. When this bit is set, it indicates that the stream is in Kernel Mode, i.e., the stream can execute privileged instructions and cannot be interrupted. When this bit is clear, it indicates that the stream is in User Mode, i.e., the stream can be interrupted and cannot execute privileged instructions. The PS also contains the Trace bit (T). When this bit is set, it indicates that the stream can be traced after executing an instruction. After each instruction, the stream traps to a Kernel Mode system routine that traces the instruction, and then returns to the trapped process for the execution of its next instruction. When the Trace bit (T) is clear, it inhibits trace traps from occurring. In addition to these bits, the PS also contains four condition code

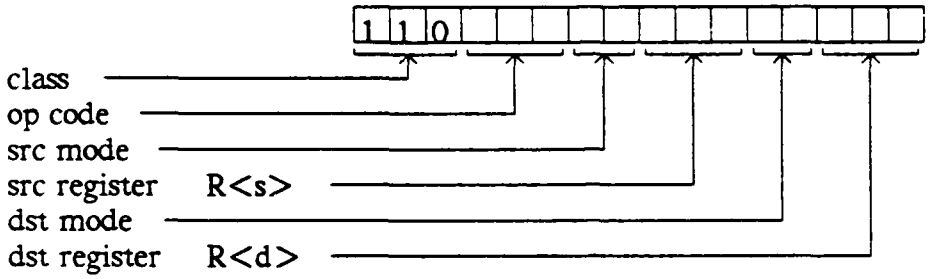


Figure 2.1 Instruction Register Format for Double Operand Instructions (DOP)

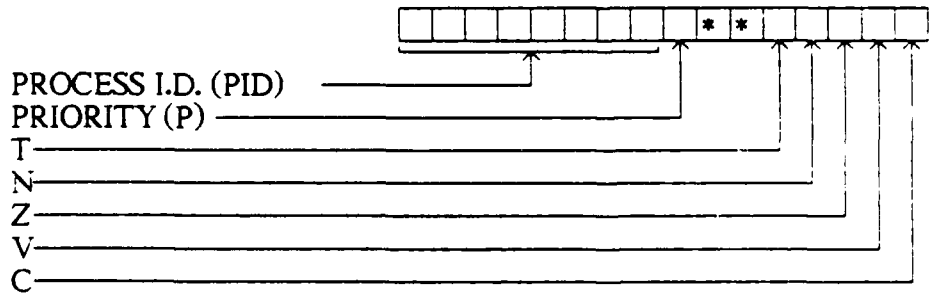


Figure 2.2 Process Status Word (PS)

flags. These are the negative (N) flag, the zero (Z) flag, the overflow (V) flag, and the carry (C) flag. These flags are set according to the results of ALU operations, and can also be set or reset by condition code instructions.

The format of the Dynamic Status Register (STAT) is shown in Figure 2.3. It contains the status of the stream as it travels through the pipeline. Five of these bits are used by the system trap instructions. These five bits are INTERRUPT, MEMERR, STOP, IIT, and TRACE. When INTERRUPT is set, it indicates that the stream is being interrupted. When MEMERR is set, it indicates that the stream is executing a memory error trap. When STOP is set, it indicates that the stream is about to be stopped. When IIT is set, it indicates that an illegal instruction was executed, or that a User Mode stream tried to execute a privileged instruction. When TRACE is set, it indicates that a trace trap is occurring. Four other bits are used to indicate that a wait cycle is occurring. In a wait cycle, the stream must pass through the rest of the cycle, executing NOP's in each segment. When the Halt Buffer Full (HBF) flag is set, the process is trying to halt another process, but must wait for the Halt Buffer (HLTB) in Segment One to become empty. The Write Request Wait (WRW) flag indicates that the Memory Write Buffer (MWB) in Segment Six was full when the stream tried to request a memory write. The Read Request Wait (RRW) flag indicates that the Memory Read Buffer (MRB) was full when the stream tried to request a memory read. The Data Wait (DW) flag indicates that the stream is waiting to receive data requested from memory. These flags are necessary for the proper execution of instructions, but do not need to be saved on the stack when the process is swapped, since these conditions must be cleared before a process can be swapped.

The Stream Identification number (SID) is three bits, which uniquely identifies each of the eight active streams. It is also used as a tag for memory requests and for data which returns from memory so that memory protection and mapping can be implemented off-chip and so that returning read data can be associated with the stream that issued the request.

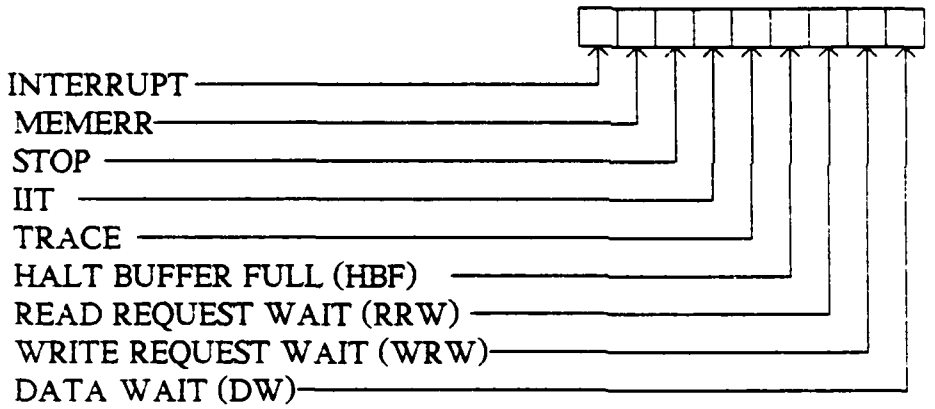


Figure 2.3 Dynamic Status Register (STAT)

The Cycle Counter (CT) is required since an instruction can take more than one pass through the pipeline to complete its execution. This counter travels with the stream and indicates which cycle of the current instruction is being executed. The Cycle Counter is set to the total number of cycles in the instruction in Segment Two on the first cycle of each instruction. The Cycle Counter is decremented in Segment Eight of the pipeline whenever the stream is not in a wait state.

2.2. Register Transfer Language

Archer [ARC82] defined a register transfer language (RTL) and used it to describe the data flow through the processor pipeline. We use the same register transfer language to describe the data flow through the modified processor pipeline. The formal language rules are not presented in this discussion, since they are given in Archer's report. An example of the use of the RTL and some of the essential terms are presented. The definitions below refer to Figure 2.4, which shows an example of a Single Operand (SOP) machine instruction.

instruction — The entire description of a machine instruction. Includes all passes through the pipeline needed to complete that particular machine instruction. Figure 2.4 describes one machine instruction.

cycle — The description of a single pass through the pipeline. The first line of each cycle has a label, i.e., a., b., ..., to identify the cycle.

microcycle — The description of a single pass through one stage of the pipeline. In the RTL description is one microcycle. NOP microcycles are omitted.

mode — Refers to one of the four addressing modes 0-3 defined in Chapter 1. Usually refers to the destination addressing mode, except that in Double Operand Instructions, a mode pair, n,m, refers to source mode n and destination mode m.

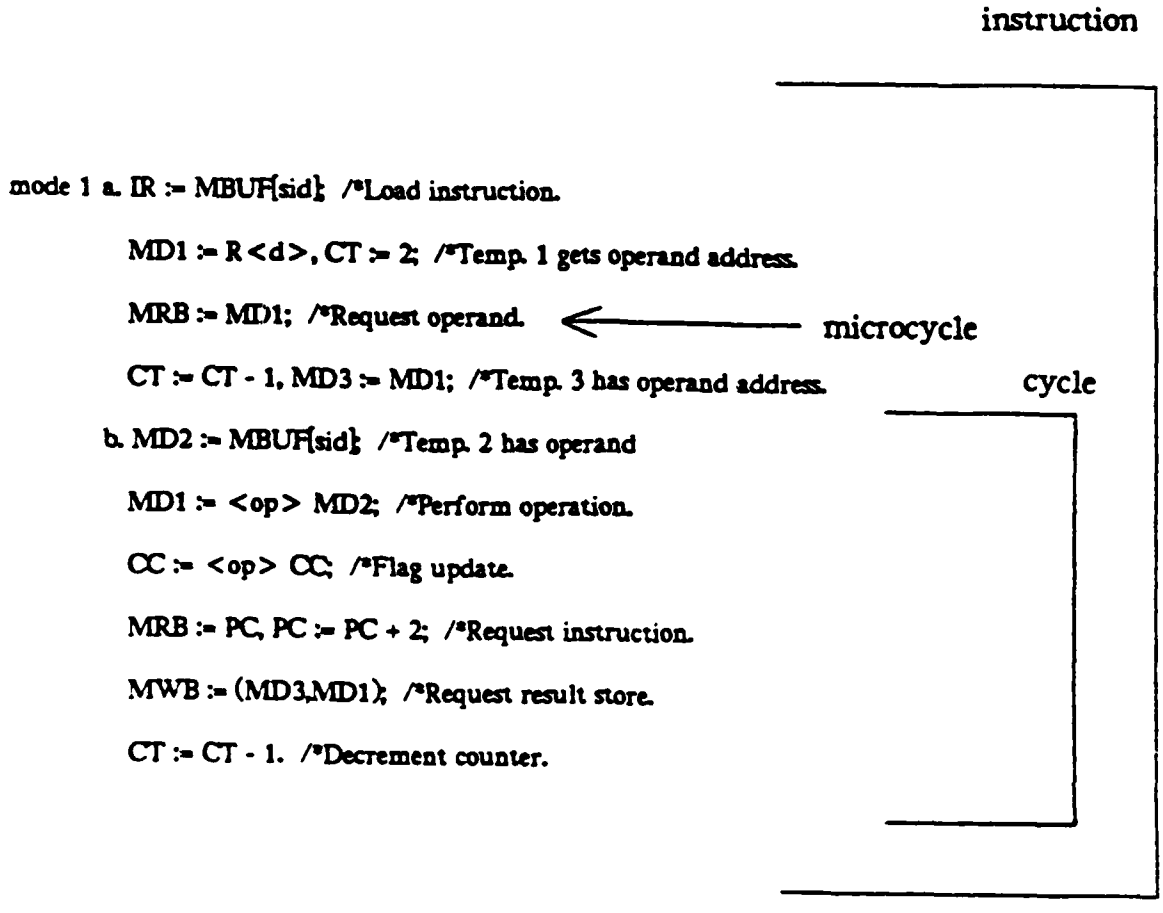


Figure 2.4 Register Transfer Language Example of SOP Instruction

register — A member of the register set or buffer set. Appendix B has a list of the registers and buffers.

function — An arithmetic or logical operation on the contents of one or two machine registers.

condition — A relation between two data elements.

conditional — A single, conditional function on one or two machine registers.

Figure 2.4 shows an example of address mode 1 of a Single Operand Instruction (memory to memory, register deferred) and illustrates the terms defined above. Mode 1 refers to register deferred mode, in which the address of the operand is in the register named in the destination field R<d>. Comments are delimited by '/*' and the end of a line. In the first cycle, cycle a., the instruction is loaded from the Memory Buffer (MBUF) into the Instruction Register (IR). Then the operand address is read from the destination register, and a memory read request is made for the operand. In the second cycle, cycle b., the operand is received from MBUF, the operation is performed, the store result request is put in the Memory Write Buffer (MWB), and the next instruction fetch request is issued.

All instructions listed in Appendix A are described in a similar way in Appendix C. The RTL routines describe the normal execution of the instructions only. The error conditions and special system traps are described at the end of the Appendix. The registers and buffers used in the description are listed and described in Appendix B.

2.3. Segment Hardware Description

This section describes the eight segments that make up the processor pipeline. Figure 2.5 shows a block diagram of the MISP processor. Segment One is the Memory Data Return Stage. In this segment, data which has been read from memory is taken from the Memory Data Buffer (MBUF). When the designated stream reaches this segment, the data is loaded into one of the temporary storage registers. Segment Two is the Register File Read Stage. In this segment, data is read from the register file.

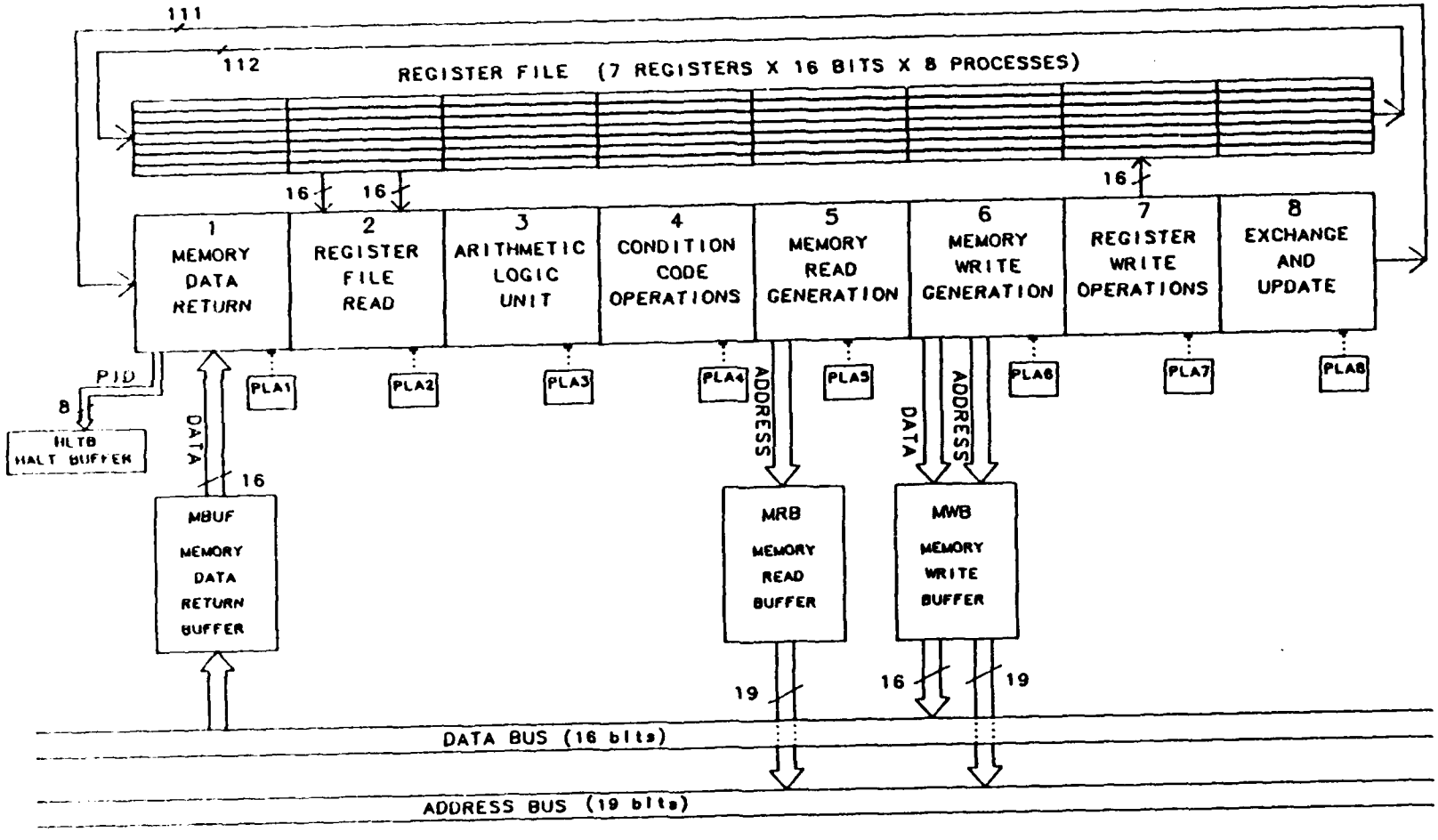


Figure 2.5 MISP Processor Block Diagram

Segment Three is the Arithmetic Logic Unit. In this segment, the arithmetic and logic operations are performed on the data. Segment Four is the Condition Code Operations Stage. In this segment, the condition flags (N,Z,V,C) are updated as a result of the previous arithmetic or logic operations. Segment Five is the Memory Read Generation Stage. Memory read requests are generated by depositing the address into the Memory Read Buffer (MRB). Segment Six is the Memory Write Generation Stage. In this segment, memory write requests are generated by depositing the address and the data to be written into the Memory Write Buffer (MWB). Segment Seven is the Register Write Operations Stage. In this segment, results are written into the register file. Segment Eight is the Register Exchange and Cycle Counter Adjustment. In this segment, data is exchanged among the temporary registers, and the pass counter is decremented.

The state of a given process is carried in the stream through the pipeline. The state consists of the data in the Program Counter (PC), the Instruction Register (IR), the three temporary storage registers (MD1-MD3), the Process Status Word (PS), the Dynamic Status Register (STAT), the Stream Identification number (SID), and the Cycle Counter (CT). These registers are shifted from segment to segment within the pipeline. Intersegment registers between segments are used for this purpose.

The processor must be bootstrapped after it is first powered up to start processes in each segment. The processsor is bootstrapped by setting the RESET pin high. The processor can be reinitialized either by setting the RESET pin high or by a Kernel Mode process executing a RESET instruction. Either of these actions will set RESETLINE, which is the output of a flip/flop that goes to each segment. The logic for RESETLINE is shown in Figure 2.6. When RESETLINE is high, all other instructions are overridden. When RESETLINE is first set, a counter is started on the rising edge of the START signal. The COUNT (3 bits) is compared with 111 (binary), reached after 8 clock cycles, and when the compare is true, RESETLINE is reset. The counter will also set the Stream I. D. (SID) appropriately for each process when it passes through Segment One. The RESET instruction is used right after turning the processor on, and the SID's need to be initialized. When RESETLINE is first set, the ERR bits for all

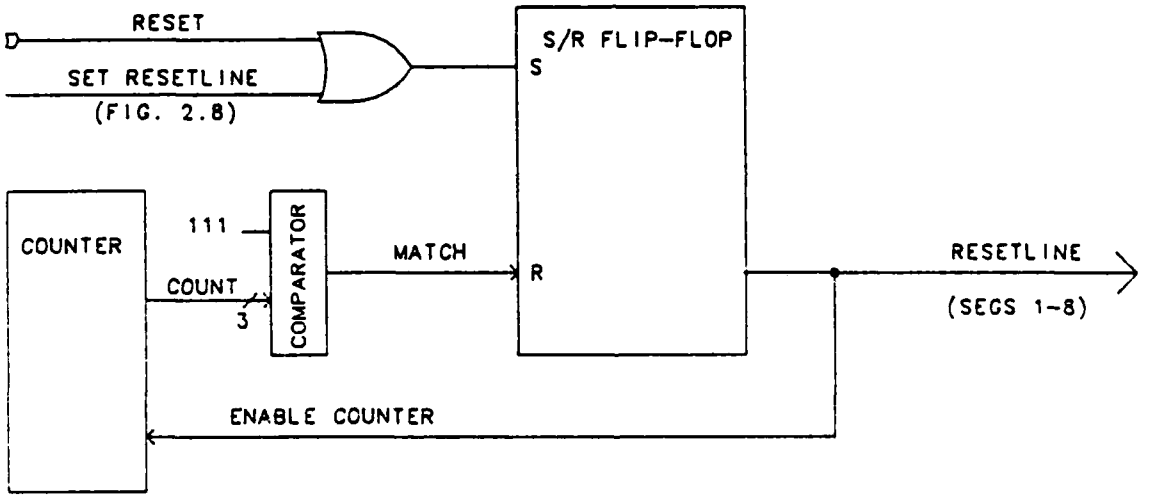


Figure 2.6 Logic for RESETLINE

streams are cleared. In addition, the FULL bits for MBUF are cleared, and all memory requests in the Memory Read Buffer (MRB) and the Memory Write Buffer (MWB) ar cancelled. These actions occur in the hardware for each buffer automatically when RESETLINE is first set. RESETLINE acts as a control signal to each of these buffers. While RESETLINE is high, each stream executes NOP's until it gets to Segment One. When the process gets to the first segment, a START instruction is loaded into its instruction register. Each stream then executes the START system trap. The Program Counter and the Process Status Word are both loaded from the trap vector located in low memory. A system trap routine is then executed to start up jobs. Chapter 3 describes this action in greater detail.

The hardware of each segment is described in the next sections. A discussion of the conventions used in the figures in these sections follows. Data flow in these diagrams proceeds from the top down. It is assumed that the data flows directly from the input intersegment registers to the output intersegment registers for all registers that are not modified by the logic shown in the segment itself. For clarity, this flow of data is not shown in the diagrams. Hollow arrows on lines indicate lines which go off-chip. Hollow tails on lines indicate lines that come from off-chip. The control for this processor is handled by individual PLA's within the eight segments, rather than by one large PLA. The control outputs for each segment and the inputs to each PLA are described in the section for each segment. Only the control outputs (indicated by dotted lines) from each PLA are shown. The inputs to the PLA are omitted from the diagrams for clarity.

Appendix D lists the conditions for the microcontrol. For each segment the microinstructions for that segment are listed along with the conditions under which that microinstruction is executed. The list includes the conditions that would be inputs to a PLA decoder, although not all of the conditions listed necessarily need to be inputs to the PLA for a particular segment. These additional conditions were listed for completeness since they represent part of the state of the process when the microinstruction is executed. The state of the Dynamic Status Register (STAT) bits is listed, but not all of these bits are needed as inputs to each segment. The inputs needed for a PLA implementation are listed and

described in the sections below. The table does not specifically list the control signals that are needed to execute the microinstructions. These control signals are listed and described in the text in the sections below. It would be a straightforward task to specify the output control signals needed to execute each microinstruction and reduce the table for a PLA implementation.

2.3.1. Segment One—Memory Data Return Stage

Segment One contains the hardware needed to load the data that has been read from memory as well as hardware that controls the Halt Buffer (HLTB). The micro-operations that are executed in Segment One are listed in Table 2.2. The operations in Group A and Group B occur in parallel, so one microinstruction consists of one operation from each group. Figure 2.7 shows a block diagram of the hardware for Segment One.

The Memory Data Buffer (MBUF) has eight locations (one location for each stream) and receives data (and instructions) that have been read from memory. The Stream I. D. (SID) returned with the data is used as a MBUF write address. Also associated with each data word is an error (ERR) bit. When the data is put into MBUF[sid] and the error bit is put into ERR[sid], the FULL[sid] bit of MBUF is set to one. The Read Data Return (RDR) signal indicates that the memory is returning previously requested data that is to be written into MBUF. Note that RDR signals and accompanying SIDs need not be synchronized with the flow of the streams in the processor pipeline; the MBUF design and SID tagging ensure proper operation.

When the stream executes a "MD2 := MBUF[sid]" microinstruction or an "IR := MBUF[sid] microinstruction, one of either TRY MD2 or TRY IR is high, respectively. If FULL[sid] is high and ERR[sid] is low, the load from MBUF into IR or MD2 is permitted. This is done by AND'ing FULL and NOT ERR with the TRY MD2 as well as the TRY IR control signals from the PLA. The output of one of these AND gates is high when the stream is expecting to load either data or a new instruction,

Table 2.2 Micro-operations for Segment One

Group A

```
IR := MBUF[sid]
IR := [Interrupt Instruction]
IR := [Halted Process Instruction], STOP := 0
IR := [Trace Trap Instruction]
IR := [Illegal Instruction Trap], STAT = 0
IR := [Memory Error Trap], MEMERR := 1, ERR[sid] := STOP :=
        IIT := TRACE := HBF := WRW := RRW := DW := 0
IR := [START Trap], STAT := 0
MD2 := MBUF[sid]
HLTB := MD2, HT := 1, HBFULL := 1
V := HT, HT := 0, HBFULL := 0, HLTB := 0
NOP
```

Group B

```
STOP := 1, HT := 0
STOP := 1, HT :=0, CT :=1
CT := 1
NOP
```

when the buffer location is full, and when no error has occurred. The result of these two AND gates is used to control MUX1D and MUX1N. MUX1D selects the new contents of IR from MBUF, from the previous contents of IR, or from the IR CODES block. MUX1N selects the new contents of MD2 from MBUF or the previous contents of MD2. In addition, either LOAD IR or LOAD MD2 is high, so CLEAR FULL is high, which clears the FULL[sid] bit in MBUF. The CLEAR FULL signal is complemented to load a 0 into the Data Wait (DW) flag in STAT. The only time that TRY IR or TRY MD2 is high while ERR is high is when the data arrived after the current stream arrived in Segment Eight, and an error had occurred. If this occurred, the PLA already generated invalid TRY IR or TRY MD2

signals, since the decode occurred one cycle before the stream arrived in Segment One. ERR is an input to this PLA but changed its value when the data arrived, which was after the decode. To assure that this data is not loaded, the ERR[sid] disables the LOAD IR or LOAD MD2 signals by forcing zeros to both AND gates, which also means that CLEAR FULL is low, and a one is gated into DW. The stream then executes a WAIT cycle, and a Memory Error Trap is forced into IR from IRCODES on the next cycle.

If a load from MBUF is being attempted, but the FULL bit is low, the data has not yet arrived in the buffer. Since FULL is low, LOAD IR and LOAD MD2 are both low, which means that CLEAR FULL is also low, and the FULL bit won't be cleared. The CLEAR FULL is then complemented, which will set the Data Wait (DW) flag. The stream then passes through the pipeline in a Wait cycle until it arrives at Segment One again, and FULL[sid] has been set. When this occurs, the stream executes the load as above.

If neither of the microinstructions "MD2 := MBUF[sid]" or "IR := MBUF[sid]" are executed, TRY MD2 and TRY IR are both low, and no load is attempted. In this case, the zero is selected to be loaded into DW. The loading of DW and the clearing of FULL[sid] are not explicitly mentioned in the Register Transfer Language description, but these actions occur automatically each time MBUF is read.

In the system trap instructions, the Instruction Register is loaded differently. It is loaded from one of six hardwired locations in IRCODES containing the opcode of the particular system trap to be executed. In addition, as can be seen from the table, when some of these microinstructions are executed, they affect some of the bits in STAT. Many of the bits have to be cleared or set independently of the rest, so these bits require their own multiplexers. Four of the bits, IIT, TRACE, WRW, and RRW, can be cleared simultaneously, so the multiplexers MUX1H and MUX1J share one control line.

In addition to the MBUF hardware, Segment One also handles the Halt Buffer (HLTB). A Kernel Mode process can halt another process by loading that process' Process I.D. (PID) into the Halt Buffer

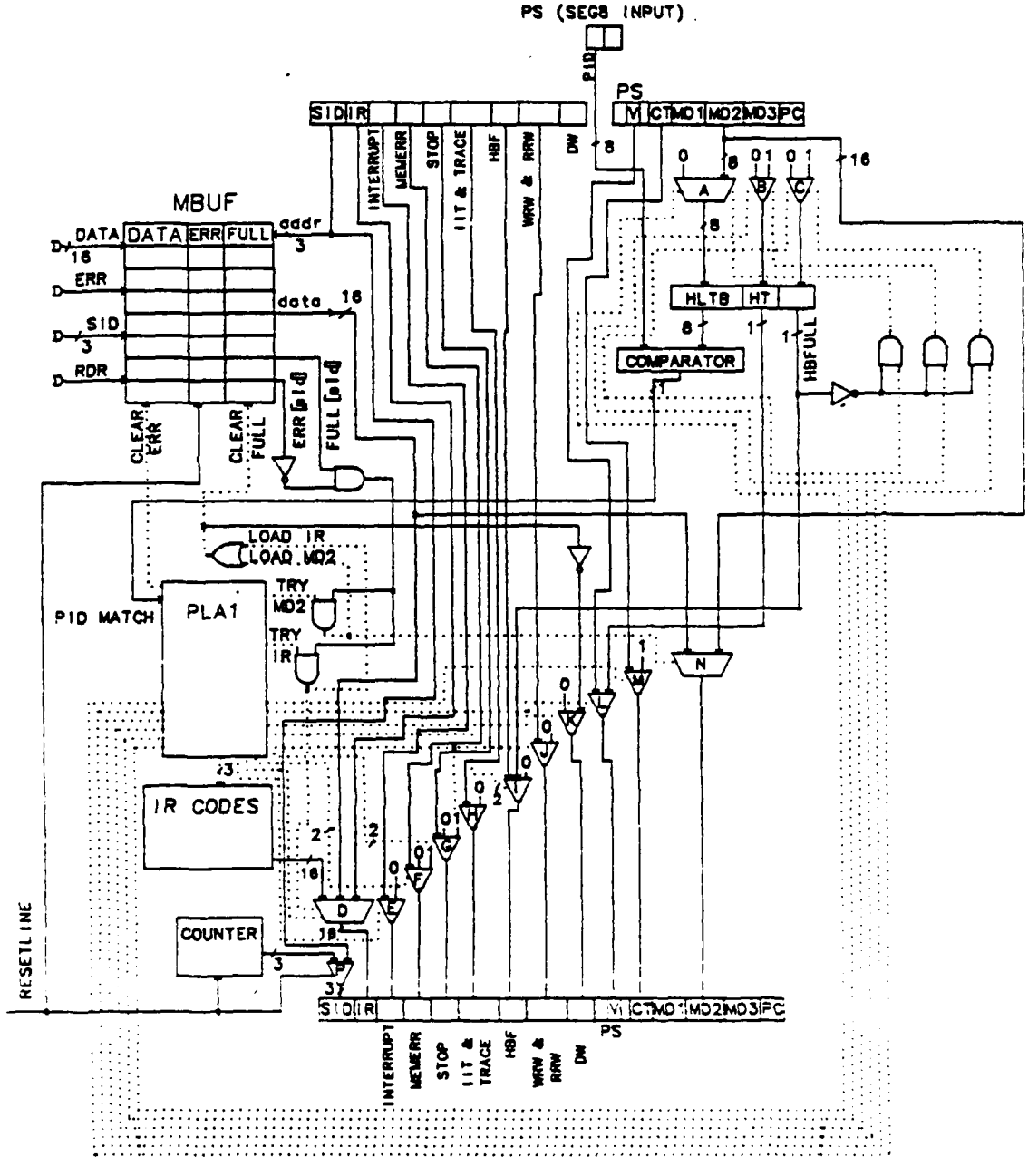


Figure 2.7 Segment One Block Diagram: Memory Data Return

(HLTB). The Halt Buffer contains eight bits for the PID, and there are two additional bits associated with the Halt Buffer. One of the bits is HBFULL, which, when set, indicates that the HLTB is currently being used to halt a process. The other bit is HT. As an example, suppose Process A is a Kernel Mode process and wants to halt Process B. The HBFULL bit is set when Process A deposits the PID of Process B into the Halt Buffer (HLTB). At the same time, HT is set. Each process, as it passes through Segment Eight, compares its PID with the PID in HLTB. The comparator shown actually needs to compare the PID of the stream in Segment Eight with the current HLTB. This comparison is necessary because the result of the compare is used as an input to the PLA, but the decoding precedes the execution by one cycle. When Process B passes through Segment Eight, the compare will be true, so Process B sets its STOP bit to indicate that it must be stopped at the end of its current instruction. Process B also clears the HT bit. This indicates that the process to be halted was found and will be halted. The next time that Process A passes through Segment One, it will load its V bit in its Process Status Word with HT. The V bit can then be checked to see whether Process B was active in some stream, and HLTB, HT, and HBFULL will each be cleared. These operations require that HLTB, HT, and HBFULL each have one multiplexer. To perform the microinstruction "HLTB := MD2, HT := 1, HBFULL := 1", the Halt Buffer must be empty. The additional logic gates shown to the right of the Halt Buffer in Figure 2.7 make sure that the Halt Buffer is empty before this microinstruction is executed.

The other microinstruction that occurs in Segment One, "CT := 1" occurs only when a process that is trying to execute a HLTP instruction identifies its own PID in the Halt Buffer. To prevent it from halting another process, the Cycle Counter (CT) is set to one. This will cause it to jump to its last cycle without waiting to deposit another PID in the Halt Buffer. It will then halt itself but it will not execute the HLTP instruction to halt some other process.

PLA1 generates the control signals for Segment One. There are twenty-three inputs to PLA1. Ten bits of the Instruction Register are used. These bits are IR[15:8] and IR[4:3]. These bits contain the instruction class and the opcode as well as the addressing modes for the source and destination. The

only bits that are not needed are IR[7:5] and IR[2:0], which contain the address of the source and destination registers and are not needed to decode the instruction. The same Instruction Register bits are used for all of the segments. Additional inputs include the RESETLINE and the Cycle Counter (CT). Seven bits of the Dynamic Status Register (STAT) are also used. These are INTERRUPT, MEMERR, STOP, IIT, TRACE, RRW, and WRW. The PID MATCH bit from the comparator is needed because of the Halt Buffer hardware. As mentioned before, this match bit applies to the stream that is in Segment Eight at the time of the decode. The ERR[sid] bit is also used. Again, this bit is really associated with the process in the previous segment when it is used as an input to the PLA. MBUF includes the necessary hardware to address the proper ERR bit when it is to be used for the PLA input.

There are twenty-five outputs from PLA1. Three bits are used to select the IR code from the hardwired IR CODES. Two control lines are needed to load and select the HLTB value. One of these lines is generated by the PLA to be AND'ed with HBFULL, and the other line is SELECT HLTB, which is a line sent directly to MUX1A to select the input. This is also true for MUX1B, which selects the data to HT, and MUX1C, which selects the data for HBFULL. So, a total of six bits are needed for the HLTB logic. One bit is used to clear ERR[sid]. One bit (TRY IR) is sent to be AND'd with the buffer logic to generate LOAD IR. LOAD IR, along with one bit sent directly from the PLA (SELECT IR), is used to select the IR at MUX1D. MUX1E needs one bit to select INTERRUPT. MUX1F needs two bits to select MEMERR. MUX1G needs two bits to select STOP. MUX1H and MUX1J use CLEAR PS, which clears five bits of STAT. MUX1I uses two bits to select HBF. MUX1K uses one bit to select DW. MUX1L uses one bit to select V. MUX1M uses one bit to load CT. The PLA sends one bit, TRY MD2, to be AND'd with other bits to generate LOAD MD2. This, along with another bit sent by the PLA (SELECT MD2) is used to select MD2.

2.3.2. Segment Two—Register File Read Stage

Segment Two contains a dual read port for the register file. It also contains the hardware necessary to set the Cycle Counter in the first cycle of a new instruction. There are also some status bit operations that occur. The micro-operations that are executed in Segment Two are listed in Table 2.3. The operations in Group A and Group B occur in parallel, so a microinstruction consists of one operation from Group A and one operation from Group B. Figure 2.8 shows a block diagram of the hardware for Segment Two.

Table 2.3 Micro-operations for Segment Two

Group A

MD1 := R<d>
MD1 := R<s>
MD1 := R6
MD2 := R<d>
MD2 := R<s>, MD1 := R<d>
MD2 := R<s>, MD1 := R6
INTLINE := 1, INTERRUPT := 0, MD1 := R6
IIT := 1
SET RESETLINE
NOP

Group B

CT := 1
CT := 2
CT := 3
CT := 4
CT := 5
NOP

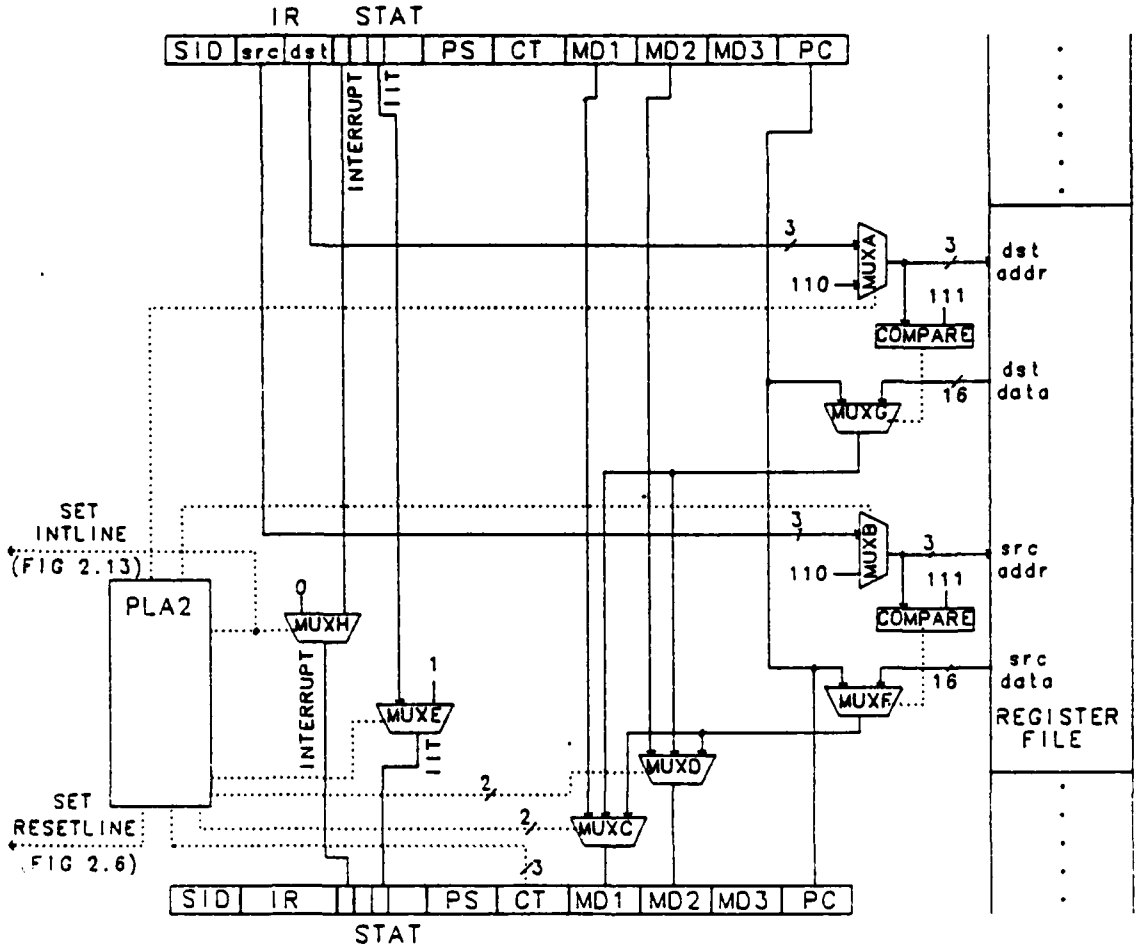


Figure 2.8 Segment Two Block Diagram: Register File Read

The register file has a dual read port which can be addressed by either the destination or the source field in the Instruction Register (IR). Registers R0-R6 are located in the register file for each stream. When the stream arrives at Segment Two, two of its registers can be read from the dual read port of the register file by gating three-bit addresses (0-6) through multiplexers MUX2A and MUX2B. The addresses can come from the Instruction Register, or a hardwired value specifying R6 can be designated by one of the microinstructions. R7, which is really the Program Counter (PC), can also be addressed. Since the PC is not in the register file, but is actually one of the registers carried through the pipeline by the stream, some additional logic is necessary to allow the Program Counter to be read when address seven is chosen. To do this, there is one comparator for the source address and one for the destination address. When a comparator sees that address seven is chosen, it generates a signal for MUX2G or MUX2F to load the destination data bus or the source data bus with the value of the Program Counter instead of the data coming from the register file. The data on these buses or the previous MD1 and MD2 values will then be loaded by MUX2C and MUX2D into MD1 or MD2, which are temporary storage registers.

The Cycle Counter needs to be set in the first cycle of each instruction when the Cycle Counter is zero. The PLA for Segment Two generates the value for the Cycle Counter. Most of the time, the PLA will simply output the value that was input to it by the Cycle Counter; however, when CT is zero, the new output is the value for CT.

There are certain conditions that cause an illegal instruction trap. An illegal instruction trap occurs when a User Mode process tries to execute a Kernel Mode instruction. A mode 0 address for the JSR and JMP instructions is also illegal. If an opcode cannot be recognized, it is an illegal instruction. Whenever an illegal instruction is detected in a stream, the IIT bit in STAT for that stream is set. This checking occurs in Segment Two. The IIT bit is then reset in Segment One when the illegal instruction trap opcode is loaded into that stream's Instruction Register.

There are other microinstructions that occur in special conditions. One microinstruction, "INTLINE := 1, INTERRUPT := 0, MD1 := R6" occurs only when a stream which is being interrupted gets a memory error. INTLINE is an interrupt signal in the processor that originates in Segment Five and causes a stream to be interrupted. When a process is interrupted, it resets INTLINE, sets INTERRUPT in its Dynamic Status Word (STAT), and handles the interrupt. If a stream gets a memory error after accepting the interrupt but before beginning to service it (i.e., while INTERRUPT is still 1), INTLINE needs to be set again so that another stream can handle the interrupt. A SET INTLINE signal is sent to Segment Five, where the flip-flop for INTLINE is located. See the section on Segment Five for a complete description of the hardware for INTLINE. A SET RESETLINE signal is used to set RESETLINE, which is shown in Figure 2.6.

PLA2 generates control signals for Segment Two. There are twenty inputs to the PLA. Ten bits of the Instruction Register (as in Segment One), as well as the Cycle Counter (CT), the PRIORITY flag, and RESETLINE are inputs. The other input bits are in STAT, the Dynamic Status Register, namely INTERRUPT, HBF, WRW, RRW, and DW.

There are twelve output bits from the PLA. MUX2A needs one bit to choose which address is needed for the source read port. MUX2B needs one bit to choose which address is needed for the destination read port. MUX2C needs two bits to control the selection of the data for MD1. MUX2D needs two bits to control the selection of the data for MD2. Three bits are sent to the Cycle Counter (CT) register. These are generated by the PLA and are determined by the current value of CT and the instruction. STIIT is sent by the PLA to control MUX2E, which sets IIT to one when STIIT is high. SET INTLINE is a signal sent by the PLA to set INTLINE in Segment Five. SET INTLINE also controls MUX2H. SET RESETLINE controls the RESETLINE as shown in Figure 2.6.

2.3.3. Segment Three—Arithmetic Logic Unit

Segment Three contains the hardware necessary to execute both single and double operand arithmetic and logic operations. These operations include shifting, addition, subtraction, and various Boolean operations. The ALU also executes some special operations that are used in trap and interrupt instructions. Table 2.4 shows a summarized list of the ALU microinstructions. Figure 2.9 shows a block diagram of the hardware for Segment Three. There are two main input data paths to the ALU itself, A and B. There is one multiplexer for each data path. The inputs to the multiplexer for A are the registers MD1 and MD2 (for normal operations), and PC and PS (for special operations). The special operations are used mostly for calculating branch and jump addresses, and pushing or popping the PC and PS in interrupt and trap operations. The inputs to the multiplexer for B are MD1, MD2, IR[7:0], IR[8:3],

Table 2.4 Microinstructions for Segment Three

MD1 := <op> MD1
MD1 := <CLEAR> MD1
MD1 := <op> MD2
MD1 := MD1 <op> MD2
MD1 := MD2 <op> MD1
MD1 := MD1 + MD2
MD1 := MD1 + 2
MD1 := MD1 - 2
MD2 := MD2 - 1
PC := MD1
PC := MD2
MD2 := PS
PC := PC + 2* IR[7:0]
PC := PC - 2* IR[8:3]
TEST MD2
NOP

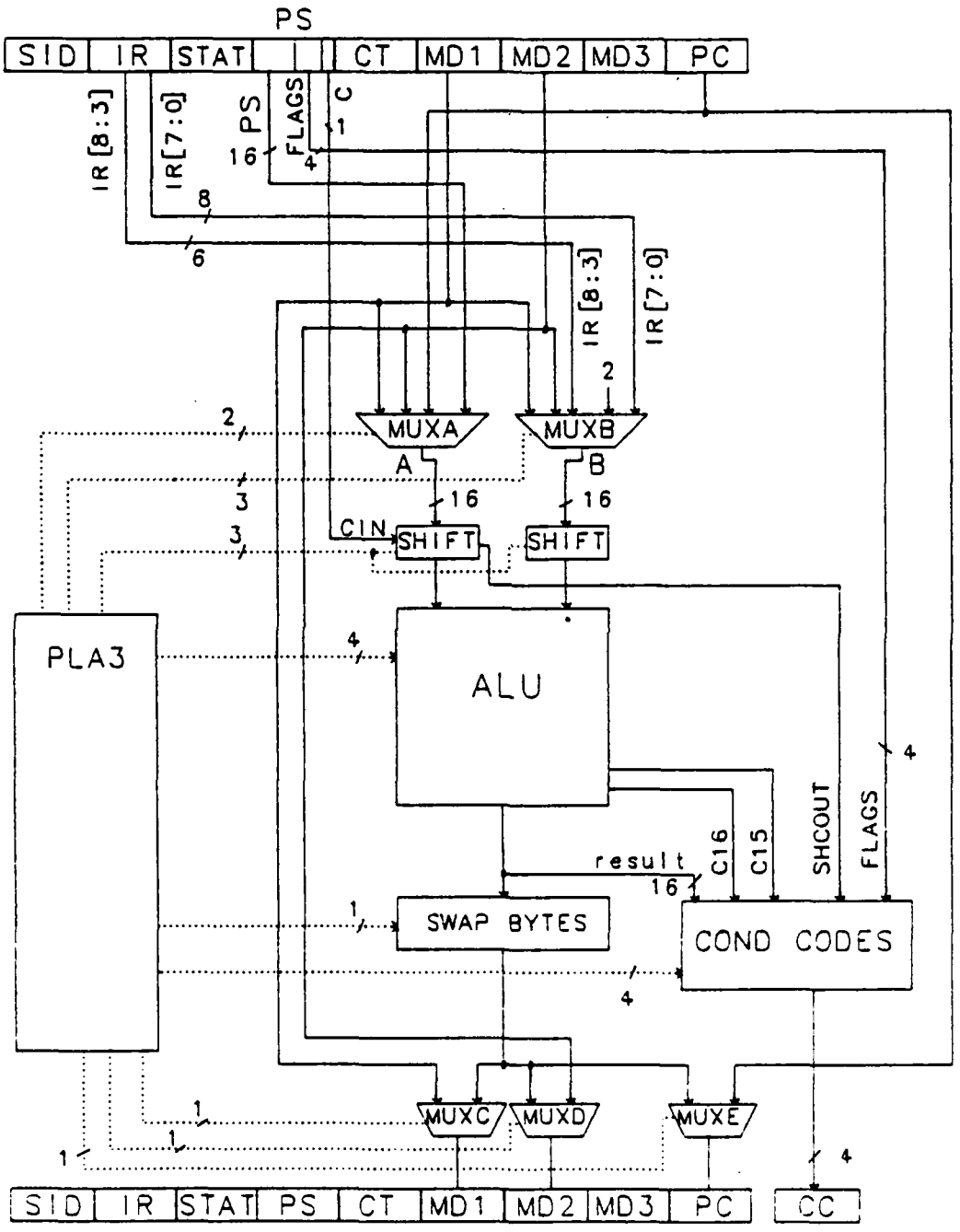


Figure 2.9 Segment Three Block Diagram: Arithmetic Logic Unit

and 2. IR[7:0] is used for branch operations, where the offset is located in the lower order eight bits. IR[8:3] is used for the SOB instruction, where the loop address is calculated from an offset located in bits 3-8. There is a shifter in each data path which can perform shift operations. The shifter for the A path also needs to perform rotate operations for the ROL and ROR Single Operand instructions. These instructions require the CIN bit which is the Carry (C) bit from the Process Status Word (PS). These shifters are used both to perform the shift and rotate operations in the PDP-11 instruction set and to multiply quantities by two, e.g., in the branch instructions, where the offset is multiplied by two. The conditions for the conditional branch instructions and the SOB instruction are checked by the PLA, which generates the control bits to perform the proper microinstruction. The "TEST MD2" microinstruction occurs in the TSET instruction and the SOB instruction. This instruction tests MD2 for a zero condition and sets the Zero flag (Z) appropriately. The ALU block performs basic arithmetic and logic operations on the A and B data paths. In the summarized table, "<op>" is used to represent any of the possible arithmetic or logic functions. These arithmetic and logic functions are listed in Table 2.5.

The results of the ALU operation are checked by the COND CODES block. The COND CODES block checks for overflow, zero result, and negative result conditions. It also selects the proper value of the carry from among the carry from the shifter, the carry from the ALU, and constants 0 and 1 as a function of the PLA outputs. Ten groups of COND CODE operations are possible. Each group is used for a particular set of instructions, and one of the ten groups is selected by the four PLA outputs sent to COND CODES. The functions for each of the ten groups are listed in Table 2.6. When the table lists "check" in the N column, it refers to checking the most significant bit of the result, and setting the N flag appropriately. When the table lists "check" in the Z column, it refers to checking the result for a zero result, and setting the Z flag appropriately. When the table lists "0" and "1" in a column, it means that the appropriate flag should be cleared or set, respectively. When the new condition codes have been determined, they are loaded into an intermediate intersegment register (CC). They are then transferred to Segment Four, where they are loaded into the Program Status Word in Segment Four.

Table 2.5 Arithmetic and Logic Operations

A + B	Add A to B
A + C	Add carry to A
A - B	Subtract B from A
B - A	Subtract A from B
A - C	Subtract carry from A
-A	Negate A
A + 1	Increment A
A - 1	Decrement A
A AND B	Logical AND
~A AND B	Clear masked bits
A OR B	Logical OR
A XOR B	Logical exclusive or
~A	Logical one's complement
A	Choose A
0	Clear result
B	Choose B

The result of the ALU block goes to the swap bytes block, which swaps the low and high order bytes of the result when enabled. This operation is used only in the SWAB instruction. The final data result can be sent to MD1, MD2, or the PC, depending on the instruction. The three multiplexers either pass the original contents of the register to the next segment or they load the result into that register.

PLA3 generates the control signals for Segment Three. There are twenty-three inputs to PLA3. Ten bits of the Instruction Register are used along with RESETLINE and the Cycle Counter. In addition, Five bits are used from STAT. These bits are IIT, HBF, WRW, RRW and DW. The four flags from the Process Status Word are also inputs to the PLA. These four flags, N, Z, V and C, are used for the conditional branches.

Table 2.6 COND CODE Block Operations

Type Instrs.	N Operation	Z Operation	V Operation	C Operation
Non-ALU opers.	No effect	No effect	No effect	No effect
Additions	Check	Check	XOR(C16, C15)	C16
Subtractions	Check	Check	XOR(C16, C15)	NOT C16
Shifts	Check	Check	XOR(new N, new C)	SHCOUT
2 operand--logical	Check	Check	0	No effect
Inc./dec.	Check	Check	XOR(C16, C15)	No effect
NEG	Check	Check	XOR(C16, C15)	NOT(new Z)
TST and CLR	Check	Check	0	0
COM	Check	Check	0	1
SWAB	Check	Hi-order byte checked	0	0
TEST MD2	No effect	Check	No effect	No effect

There are twenty outputs from the PLA. MUX3A needs two bits to select the A operand. MUX3B needs three bits to select the B operand. There are two control bits indicating which one of four possible shift operations (rotate left, shift left, rotate right, and shift right) is occurring. These go to both of the shifters. There is one additional bit to enable the proper shifter. This bit also goes to both shifters. Choosing the proper ALU operation requires four bits. Choosing the proper operation on the condition codes requires four additional bits. There is one bit to enable SWAP BYTES. MUX3C, MUX3D, and MUX3E need one bit each to choose between their two inputs.

2.3.4. Segment Four—Condition Code Operations

Segment Four handles operations for the Process Status Word (PS). It updates the condition codes from the previous operation. It also sets the priority flag in PS. The microinstructions that are executed in Segment Four are listed in Table 2.7. Figure 2.10 shows a block diagram of the hardware for Segment Four.

MUX4A loads the four condition code flags from one of four places. They could come from the intermediate intersegment condition code register that has the updated flags from Segment Three. They could also come from the four lower order bits of MD2, when the microinstruction "PS := MD2" is executed. They could be transferred directly from the original flags in PS. The fourth choice is the CCBLOCK, when the microinstruction "PS[code] := IR[4]" is executed for the Condition Code manipulating instructions. The logic for this block is shown in Figure 2.11. The inputs for this block are the four condition codes from the Process Status Word (PS[3:0]) and the five lower order bits from the Instruction Register (IR[4:0]). Bit four of the Instruction Register, IR[4], contains either a one or a zero, depending on whether the flags are to be set or cleared. The four lower order bits (IR[3:0]) represent a

Table 2.7 Microinstructions for Segment Four

CC := <op> CC
PS[code] := IR[4]
PS := MD2
PS := MD2, PRIORITY := 1
PRIORITY := 0
PRIORITY := 1
UPDATE Z
NOP

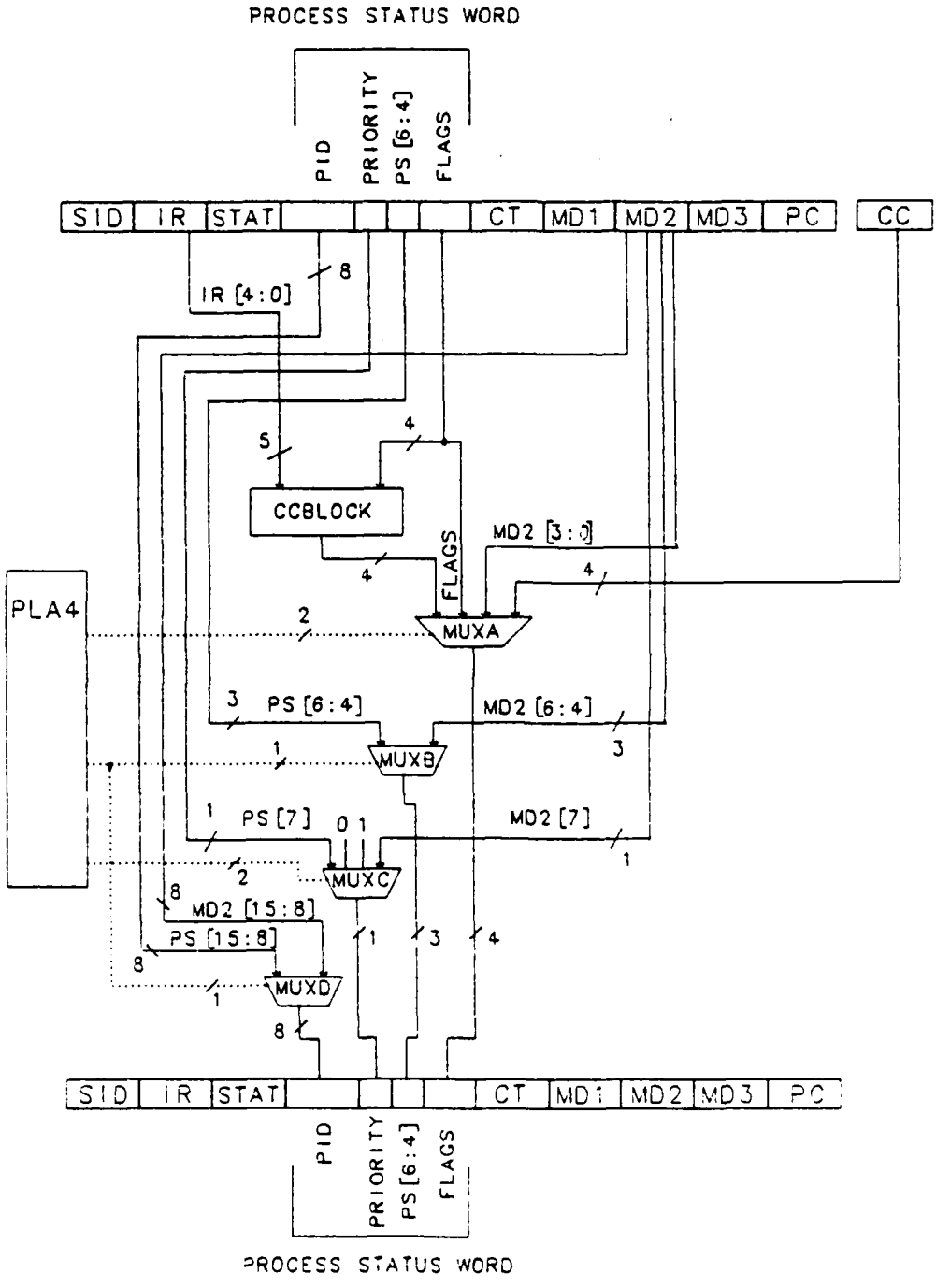


Figure 2.10 Segment Four Block Diagram: Condition Code Operations

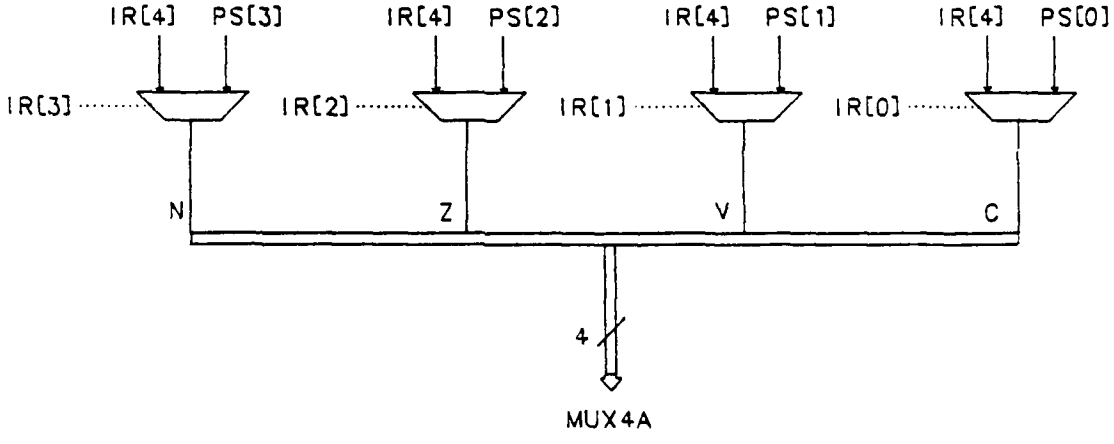


Figure 2.11 Logic for CCBLOCK

mask for the condition code bits, indicating which of the condition code bits should be set or reset. The value of each new condition code bit is selected by a multiplexer. The inputs of the multiplexer are the old value of that condition code and bit four of the Instruction Register. The mask bit corresponding to the bit position for that particular condition code selects the new value of the condition code, from either the old value of that condition code, or IR[4]. As an example, for the Z flag, if the mask bit (IR[2]) is one, the new value for Z will be the value of IR[4]. If the mask bit is zero, the value of Z will remain the same as it was, i.e., PS[2].

MUX4B loads bits 4-6 from the original PS, or from MD2 (for the "PS := MD2" microinstruction). MUX4C either sets, clears, transfers the original, or loads the Priority bit from MD2. The Priority bit is often set during System Trap instructions. MUX4D loads the Process I.D. from MD2 or preserves its value from the input PS.

There are nineteen inputs to PLA4. There are ten bits from the Instruction Register, RESETLINE, CT, and the IIT, HBF, WRW, RRW, and DW bits from STAT. Most of the bits in STAT are used to determine whether there is a NOP. The PLA generates five control signals for Segment Four. MUX4A and MUX4C each require two separate control signals. MUX4B and MUX4D share the same control signal.

2.3.5. Segment Five--Memory Read Generation

Segment Five generates read requests for memory. The address and Stream I.D. for the location are put in the Memory Read Buffer (MRB). The address and Stream I.D. are then sent from the MRB to memory. The data in the addressed location is fetched from memory and returned to the Memory Data Buffer (MBUF) in Segment One. The microinstructions that are executed in Segment Five are listed in Table 2.8. Figure 2.12 shows a block diagram of the hardware for Segment Five.

Table 2.8 Microinstructions for Segment Five

MRB := PC, PC := PC + 2
MRB := PC, PC := PC + 2, INTERRUPT := 0
MRB := PC, PC := PC + 2, TRACE := 0
MRB := PC, PC := PC + 2, MEMERR := 0
MRB := PC
MRB := MD1
MRB := MD1, MD1 := MD1 + 2
MRB := MD2
MRB := IR[7:0]
MRB := IR[7:0], IR := IR - 2
INTERRUPT := 1, INTLINE := 0
TRACE := 1
NOP

The Memory Read Buffer (MRB) is a first-in-first-out buffer with up to eight locations. If all eight locations are implemented, MRB will never be full since no stream can have more than one pending read request. In this case, all logic related to MRB FULL could be eliminated. Each location contains a sixteen-bit address word plus a three-bit Stream I.D. MRB is loaded from MUX5A whenever the ENRB signal is high. ENRB is high when the PLA sends out a signal to execute a microinstruction that makes a read request, and when the MRB is not full. ENRB gates the SID and the address into a location in the MRB. MUX5A chooses the address from MD1, MD2, the Program Counter, or the lower eight bits of the Instruction Register filled with eight high-order zeros. MD1 and MD2 are usually used when data is being requested. The Program Counter is used when the instruction fetch is being executed. The low-order byte of the Instruction Register is used when a trap is being executed, since the low-order byte contains the second address of the two-word trap vector in memory which contains the new Process Status Word. The new Program Counter is located in the first word of the trap vector and is

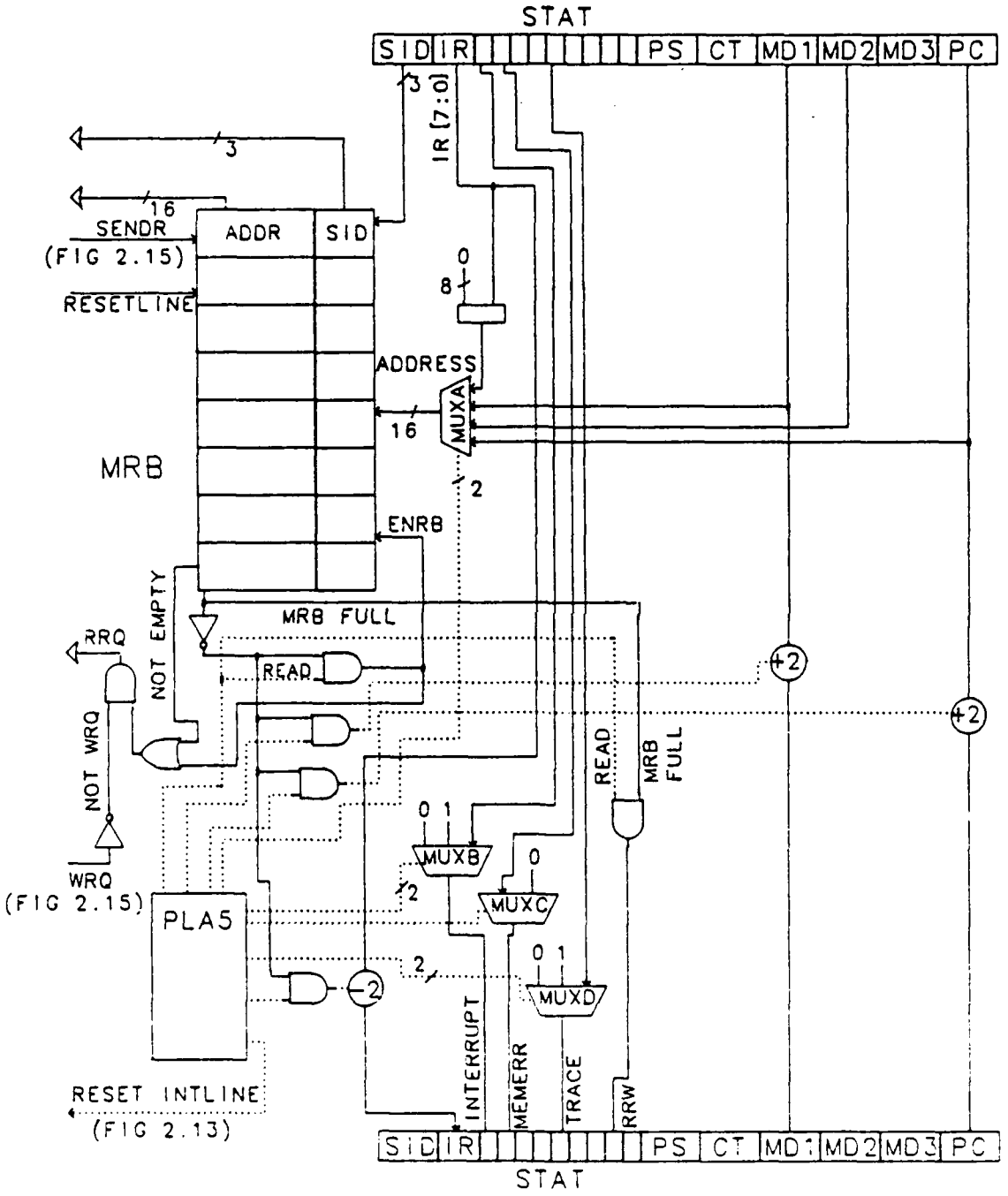


Figure 2.12 Segment Five Block Diagram: Memory Read Generation

addressed by the IR after it is decremented by two. Trap operations are described in greater detail in Chapter 3.

To issue a read request, there is one line sent out to the memory called Read Request (RRQ). RRQ is generated whenever the NOT EMPTY flag from the MRB or the signal ENRB is high which indicates that a request is about to be put in the buffer, and there is no write request being generated. The Memory Write Buffer has priority over the Memory Read Buffer, and no read requests are made unless the MWB is empty (no WRQ). SENDR is a signal generated in Segment Six from the Bus Available signals. SENDR gates the address out onto the bus. SENDR is high whenever the Address Bus Available (ABA) signal is high and RRQ is high. A diagram of the logic for these signals is included in the section describing Segment Six.

The Read Request Wait (RRW) flag is set by logic in this segment. If the Memory Read Buffer is full when the stream tries to execute a memory read request, the output of the AND gate (MRB FULL and READ) will be high, which will cause the RRW flag to be high. The stream will then execute wait cycles until the MRB is not full.

There are some microinstructions which increment the Program Counter or MD1 after the read request is made. The Program Counter is incremented to point to the next instruction. MD1 is incremented when it is being used in autoincrement mode. That incremented value is subsequently put in the source or destination register. There is also a decrementer which decrements the Instruction Register. The lower byte of the Instruction Register points to the trap vector. In the trap instructions, the new Program Status Word is loaded with the value located at the address pointed to by the lower byte of the IR. In this case, the new Program Counter will be loaded in the next cycle, from an address two less than the first address. The decrementer is used in this cycle to form this lower address. Since the memory is addressed in bytes, but the desired locations are on word boundaries, it is necessary to either increment or decrement by two in each case. Since these registers are not always modified, the incrementers and decrementer can be engaged or disengaged.

This segment also modifies the MEMERR, TRACE, and INTERRUPT bits of the Dynamic Status Word (STAT). MEMERR is cleared on the last cycle of the Memory Error trap instruction. TRACE is set if the stream is about to be traced. The stream will be traced if the T bit in PS is set, and if no other system traps are about to occur. TRACE is cleared on the last cycle of the Trace trap instruction.

Segment Five also controls some of the interrupt logic. INTLINE is the output of a flip-flop that is set by INT, the interrupt line that comes from off-chip. The logic involved with setting and resetting INTLINE is shown in Figure 2.13. The INT pin is pulsed by the off-chip interrupt controller for one clock period. In that time, the flip-flop will have been clocked, so the output of the flip-flop, INTLINE, will be set. In Segment Five, INTLINE is an input to the PLA. If INTLINE is high, the microinstruction "INTERRUPT := 1, INTLINE := 0" will be executed if the stream is not in Kernel Mode, and it can be interrupted at that time. INTERRUPT is the bit in the Dynamic Status Register (STAT), which indicates that the stream is about to be interrupted. The PLA generates a control line, RESET INTLINE, which will reset the flip-flop if INT is not being pulsed again and Segment Two is not trying to set INTLINE. SET INTLINE is generated by Segment Two when the stream that was handling the interrupt has gotten a memory error. SET INTLINE is generated so that another stream will handle the interrupt that it was about to handle.

PLA5 generates the control signals necessary to execute the microinstructions in Segment Five. There are twenty-six inputs to the PLA in Segment Five. The usual ten bits of IR are used, along with RESETLINE, INTLINE and CT. Eight bits of STAT are used. These are INTERRUPT, MEMERR, STOP, IIT, TRACE, HBF, WRW, and DW. The Priority bit is used to determine whether or not a stream can be interrupted. The T bit is used to determine whether or not a stream should be traced. The Z bit is used because there is a conditional statement in the SOB instruction which depends on the Z bit.

There are twelve outputs for the PLA. The outputs of the PLA are listed below. The PLA generates the READ signal, which indicates that a read request will be attempted in this cycle. This is the signal which is AND'ed with the NOT MRB FULL signal to generate the ENRB signal which gates the

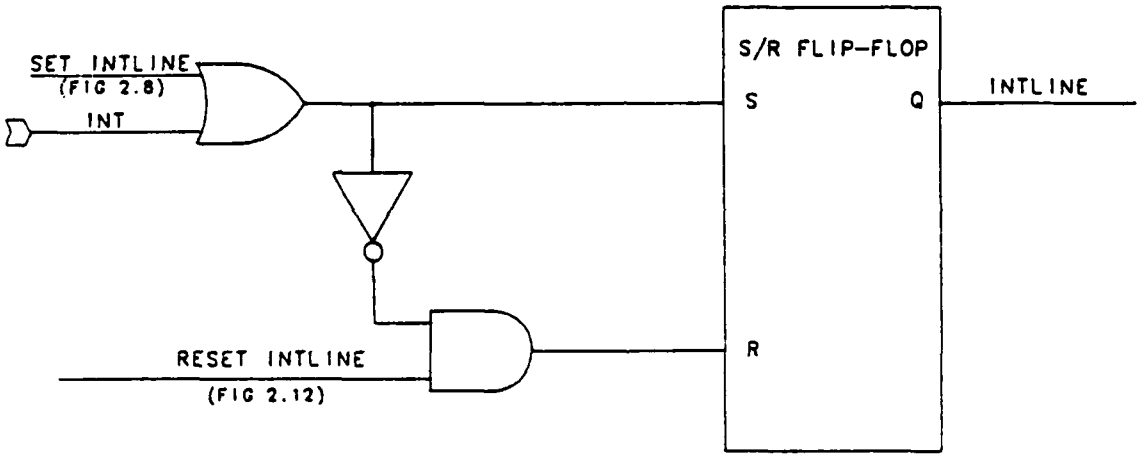


Figure 2.13 Interrupt Logic

request into the buffer. READ is also that is AND'ed with MRB FULL to set RRW. There are two lines which control MUX5A to select the address. Two lines control MUX5B, which selects the INTERRUPT bit. One line controls MUX5C, which selects the MEMERR bit. Two lines control MUX5D, which selects the TRACE bit. There are three lines which are AND'ed with NOT MRB FULL to control the incrementers and the decrementer. There is one line which is generated by the PLA to reset INTLINE.

2.3.6. Segment Six—Memory Write Generation

Segment Six generates write requests to memory. The Memory Write Buffer (MWB) is loaded with write requests consisting of the address, the Stream I.D., and the data. The MWB sends the request to memory when it receives access to both the data and the address buses. The microinstructions that are executed in Segment Six are listed in Table 2.9. Figure 2.14 shows a block diagram for Segment Six.

The Memory Write Buffer (MWB) is a first-in-first-out buffer with eight locations. Each location contains a sixteen-bit address, a three-bit Stream I.D., and a sixteen-bit data word. MUX6A is an address multiplexer which loads the address into the MWB from one of two locations. The address can either be loaded from MD1 or MD3, which are temporary registers holding the address of the memory

---

Table 2.9 Microinstructions for Segment Six

MWB := (MD3,MD1)
MWB := (MD1,MD2)
MWB := (MD1,PS)
MWB := (MD1,PC)

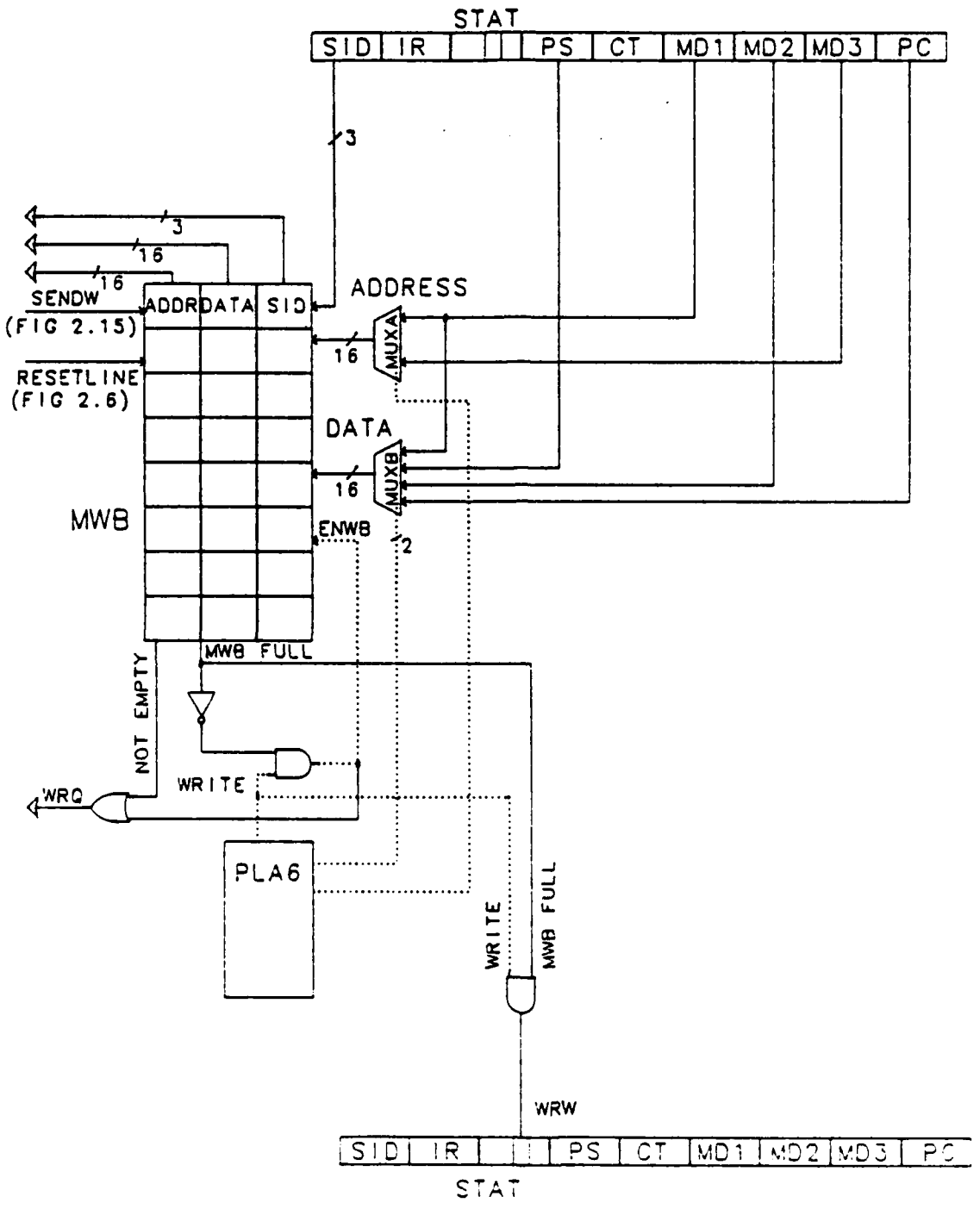


Figure 2.14 Segment Six Block Diagram: Memory Write Generation

location to be used in this operation. MUX6B is a data multiplexer which loads the data from one of four locations. The data can be loaded from MD1, MD2, the Process Status Word (PS), or the Program Counter (PC). MD1 and MD2 are temporary storage registers that are used to store data in many of the instructions. PS is used when it is necessary to push the Program Status Word on the stack during traps or interrupts. PC is used when the current Program Counter is being pushed on the stack during traps and interrupts. The ENWB signal is high whenever data is to be put into the MWB. This signal is only high when the PLA sends out a WRITE signal to generate a write request and the buffer is not full. If the buffer is full, the MWB FULL signal will be high. MWB FULL is AND'ed with the WRITE flag to set the WRW flags whenever the MWB is full and cannot accept an attempted write request. When WRW is set, it indicates that the stream must execute wait cycles until the write buffer is no longer full.

Figure 2.15 shows the logic which is used to control the access to the address and data buses. The write request (WRQ) signal is sent out on the bus to memory. It is generated whenever the MWB is not empty, or a new request is being gated into MWB. MWB NOT EMPTY is generated by the buffer when it has at least one request in it. The read request (RRQ) signal is generated in Segment Five whenever there is no write request, and either the MRB is not empty, or a request is about to be put in.

Data Bus Available (DBA) is sent to the chip when the off-chip bus controller gives the processor access to the data bus. The Address Bus Available (ABA) signal is sent to the chip when the bus controller gives the processor access to the address bus. If both the Data Bus Available and the Address Bus Available signals are received while WRQ is high, the SENDW signal will be set high. This signal gates the address (nineteen bits including the SID) and the data to the bus. If either of the buses are not available, no write transaction will be sent. Read requests will be sent out on the bus only when no write request is made, since write requests have priority over read requests. When RRQ is high, and the Address Bus Available (ABA) signal is high, SENDR will be sent to the MRB in Segment Five, which will gate the address for the read (with SID) onto the bus. Note that streams continue execution once a

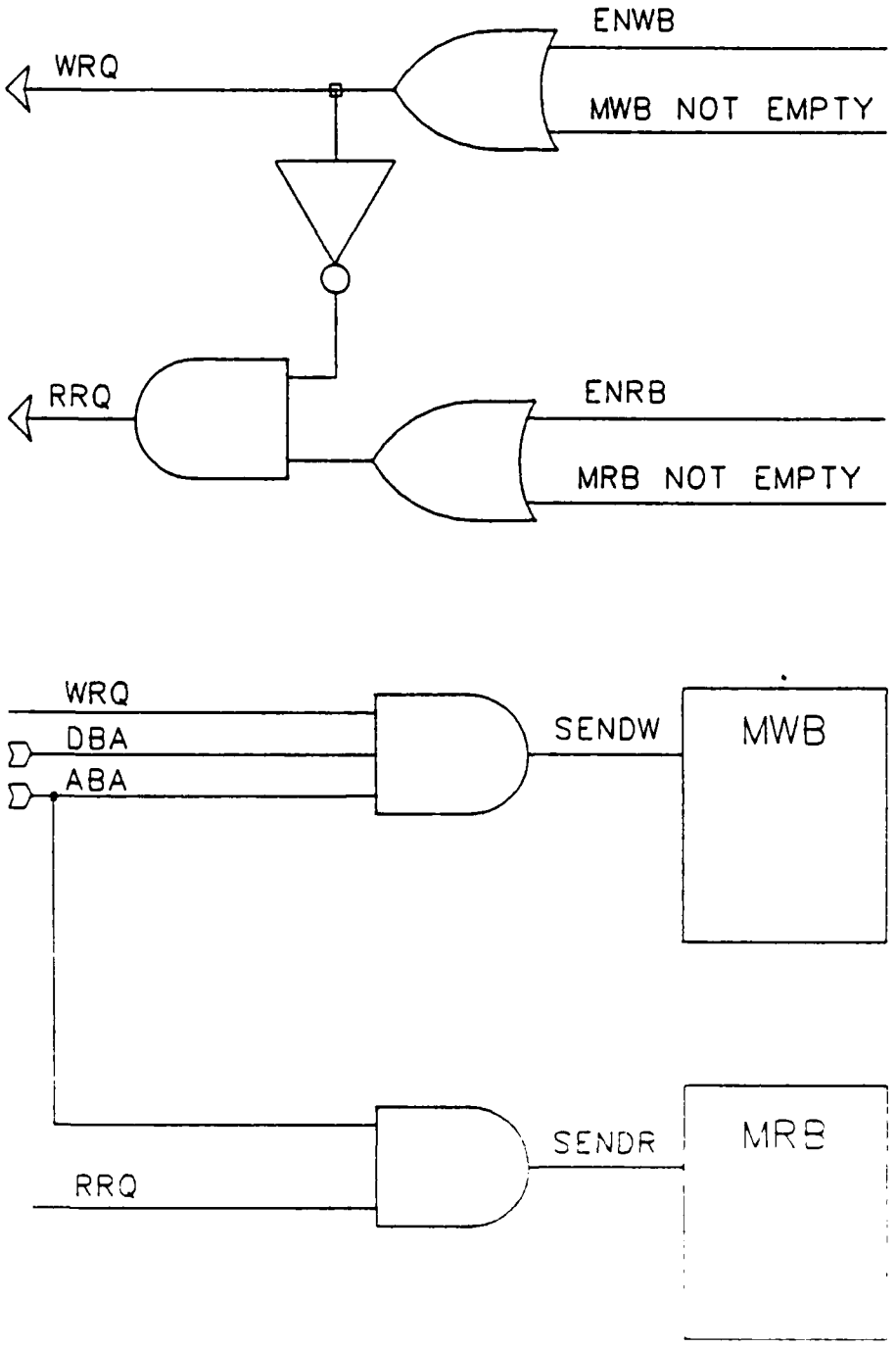


Figure 2.15 Data and Address Bus Control Logic

write request is placed in the MWB. Thus MWB can be full when a stream attempts to enter a new request. Although memory requests can proceed out of order, determinacy of execution is guaranteed by giving writes priority over reads, executing writes in FIFO order, not allowing a stream to make new requests while waiting for read data, and protecting interstream critical sections of code with programmed semaphores (see Chapter 3).

PLA6 generates the control signals for Segment Six. There are eighteen inputs to the PLA. Ten bits of the Instruction Register are used, along with the Cycle Counter. In addition, RESETLINE and four bits of STAT are used. These four bits are IIT, HBF, RRW, and DW.

There are only four outputs from the PLA. One bit controls MUX6A, which chooses the address. Two bits control MUX6B, which chooses the data word. WRITE is a signal sent out to indicate that a write request is being attempted.

2.3.7. Segment Seven—Register Write Operations

Segment Seven updates the register file after register operations have been performed. Segment Seven contains a single write port to the register file. The microinstructions that are executed in Segment Seven are listed in Table 2.10. Figure 2.16 shows a block diagram for Segment Seven.

It is possible to write to either the source or destination registers specified in the instruction. It is useful to be able to update the source register during autoincrement mode. MUX7A selects the write address from the source field in IR, the destination field in IR, or register six, which is the stack pointer. It is possible to update any register R0 through R6 when that register's address is in the source or destination field. When R7 is chosen, the comparator output will be true, and instead, the write data will be put into the Program Counter, which is R7. The data is selected by MUX7B, which chooses between MD1 and MD2. The register file also has a REG ENABLE line which gates the address and the data into the file when there is a WRITE REG signal from the PLA and the address is not seven.

Table 2.10 Microinstructions for Segment Seven

R<d> := MD1
R<d> := MD2
R<s> := MD1
R<s> := MD2
R6 := MD1
NOP

PLA7 generates the control signals necessary to control Segment Seven. There are nineteen inputs to the PLA. Ten bits of the Instruction Register, RESETLINE, and the Cycle Counter are inputs to the PLA. Additional inputs are IIT, HBF, RRW, WRW, and DW, all of which are in STAT, the Dynamic Status Register.

There are only four output bits from the PLA. Two of the bits control MUX7A, which chooses the address from the source field, the destination field, or the address six. One bit chooses the data from MD1 or MD2 with MUX7B. When the address is seven, the comparator generates a control signal to control MUX7C and causes it to gate the data to the PC. When the compare fails and WRITE REG is true, the REG ENABLE signal gates the data and the address into the register file. WRITE REG is a signal sent by the PLA when a register is to be written.

2.3.8. Segment Eight—Register Exchange and Cycle Counter Decrement

Segment Eight decrements the Cycle Counter. It also exchanges some of the registers to allow operations to be completed in fewer cycles. The micro-operations that are executed in Segment Eight are listed in Table 2.11. A microinstruction consists of one operation from Group A and one operation

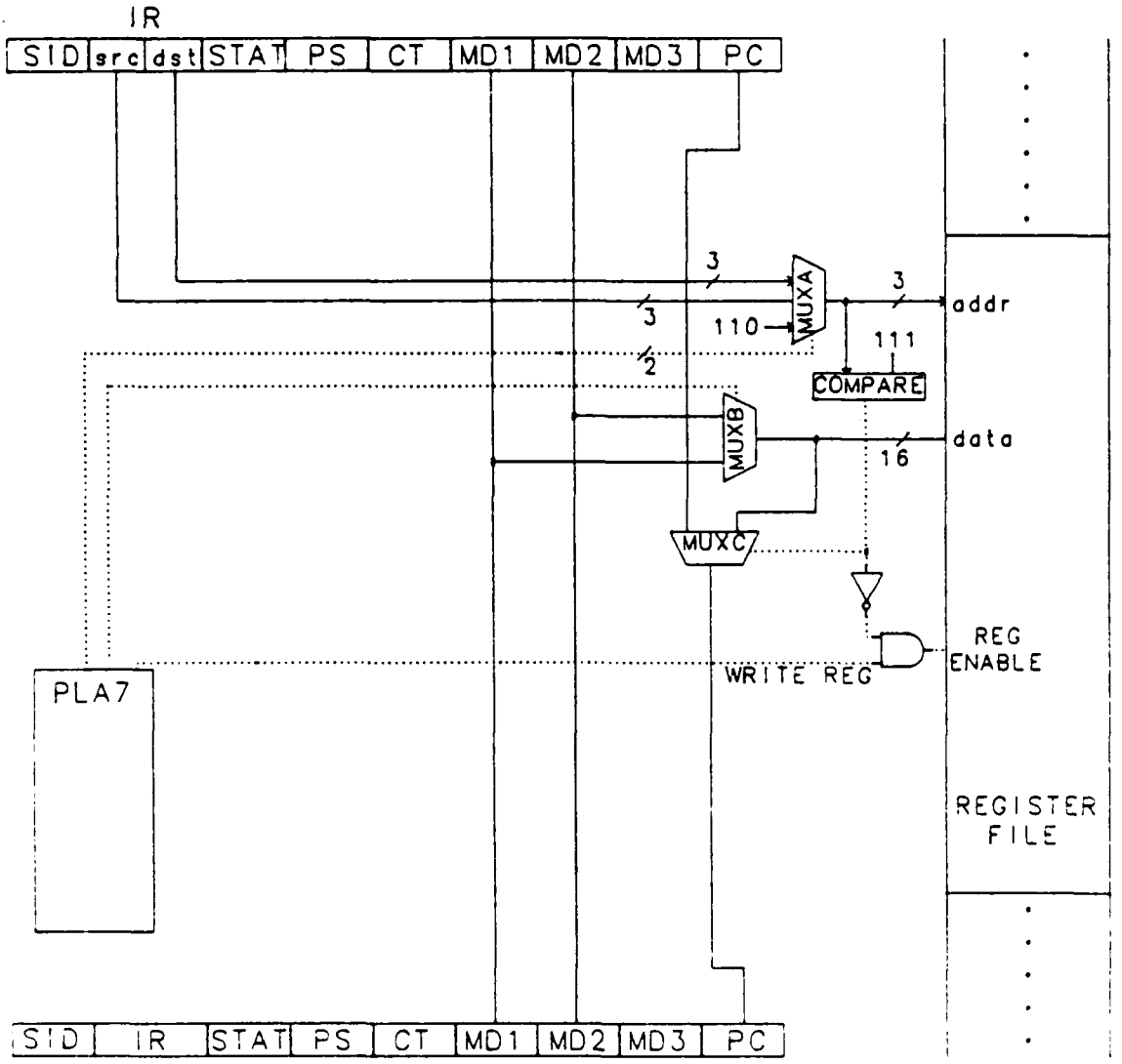


Figure 2.16 Segment Seven Block Diagram: Register Write Operations

Table 2.11 Micro-operations for Segment Eight

Group A

```
MD3 := MD1
MD1 := MD3
MD1 := MD2
MD3 := MD2
MD2 := PC
MD3 := MD1, MD1 := MD2
MD3 := MD1, MD1 := MD3
NOP;
```

Group B

```
CT := CT - 1
NOP
```

from Group B. Figure 2.17 shows a block diagram for Segment Eight.

Segment Eight contains the decrementer, which decrements the Cycle Counter. The decrement does not always occur, so it is enabled with a control line. There are three multiplexers in Segment Eight–MUX8A, MUX8B, and MUX8C. These are used for the register exchange operations. MUX8A is used to select MD2, MD3, or the old contents of MD1 to go into MD1. MUX8B is used to select either the Program Counter or the original contents of MD2 to go into MD2. MUX8C is used to select MD2, MD1, or the original contents of MD3 to go into MD3. Using these multiplexers, it is possible to exchange two registers simultaneously, as the microinstructions indicate.

PLA8 generates the control signals for Segment Eight. There are twenty inputs for the PLA. Ten bits of the Instruction Register, the Cycle Counter, and RESETLINE are used as inputs. In addition, IIT,

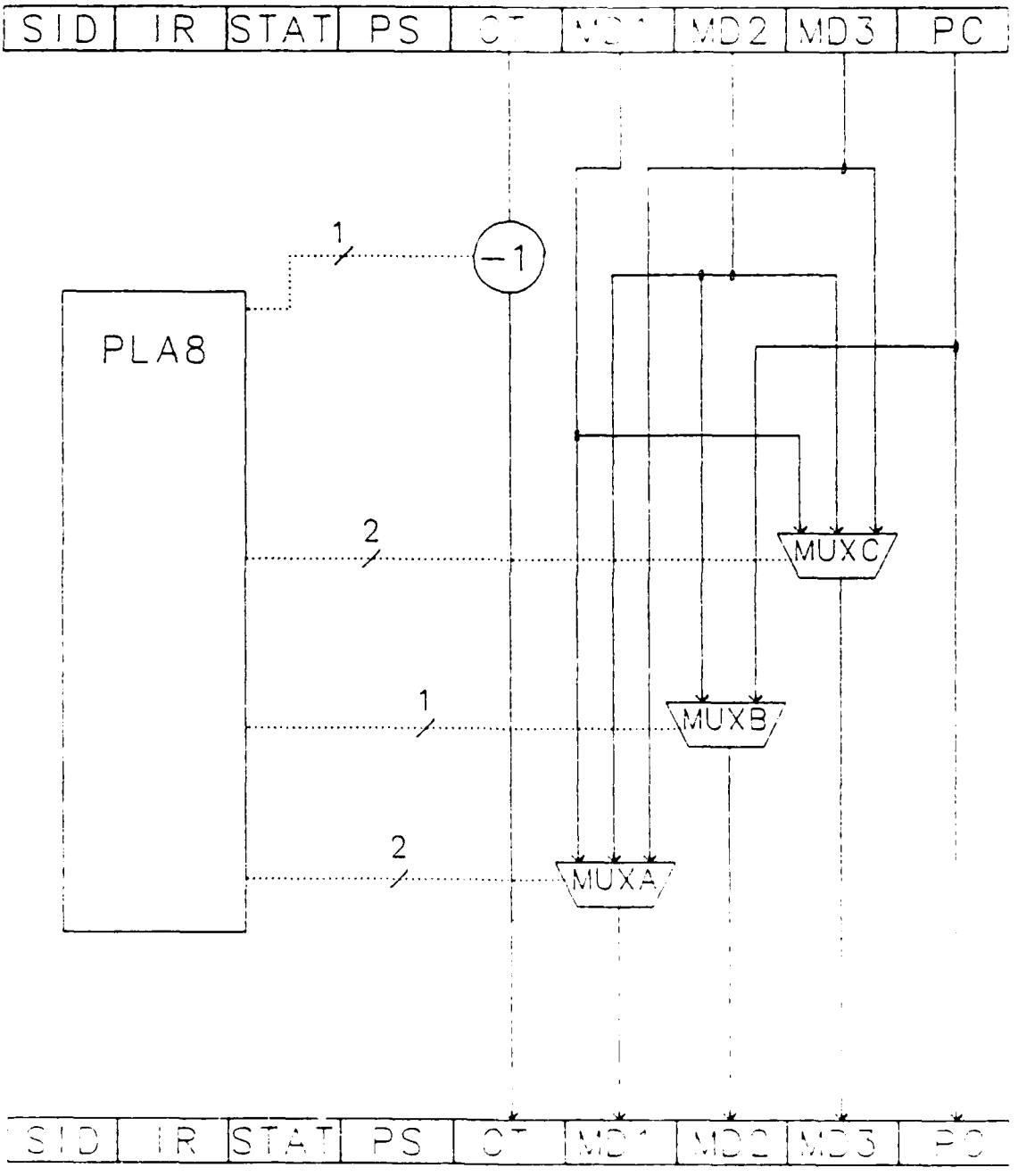


Figure 2.17 Segment Eight Block Diagram: Exchange and Update

HBF, WRW, RRW, and DW, all of which are in STAT, are also inputs. The ZERO flag in the Process Status Word is also an input to the PLA. This flag is used in the TSET instruction, in which the register is checked to see whether the semaphore was zero.

There are six outputs from the PLA. MUX8A uses two select lines to determine which register should be switched to MD1. MUX8B needs one line to choose between the PC and the original contents for MD2. MUX8C needs two lines to choose the new contents of MD3. One line is needed to enable the decrementer for the Cycle Counter.

3. MISP OPERATING SYSTEM

3.1. Operating System

The MISP processor is intended for a highly concurrent multiprocessing environment. To minimize additional hardware, much of the scheduling and processor interaction are handled by the operating system. A mechanism was developed for handling traps and interrupts. Several instructions were added to the original PDP-11 instruction set to facilitate the process control functions. The processor architecture accommodates these instructions with a minimum of additional hardware. These features of the operating system are described in the following sections, and an example of a complete system configuration with the MISP processor chip follows.

3.1.1. System Overview

The operating system primarily consists of system software routines located in low memory. (These system software routines can be protected by the memory controller.) These system software routines perform various functions needed for maintaining the multiprocessing environment. Any one or more of the eight processes could be running operating system routines at any time. Processes may have two levels of priority – Kernel Mode and User Mode. The priority of the process is designated in the Priority bit of the Process Status Word (PS). The Priority bit is one for Kernel Mode processes and zero for User Mode processes. Kernel Mode processes may execute privileged instructions. In addition, Kernel Mode processes may not be interrupted. When a process is performing an operating system function, it is usually in Kernel Mode. User Mode processes may not execute the Process Control instructions (see Appendix A) which require the Priority bit to be one. User Mode processes may be interrupted between instructions.

The operating system performs several functions. The main function is to handle process swapping and scheduling. Process swaps are necessary for the highly concurrent multiprogramming environment. The operating system loads processes into the pipeline and swaps them out when they are finished. Although only eight processes are active on the pipeline at any one time, there could be many other processes in queues (in memory) waiting to be executed. The eight-bit Process I. D. (PID) allows 256 processes to be uniquely identified. Another function of the operating system is to handle traps and interrupts. Traps are simply jumps to special operating system routines through a trap vector which contains new values for the Process Status Word (PS) and the Program Counter (PC). These traps perform various functions such as handling memory errors and illegal instructions, tracing processes, and removing halted processes. The operating system handles interrupts through interrupt service routines. A third function of the operating system is to handle interprocess interactions. This support function includes control of the semaphore registers and the halting of other processes. These operating system functions all have special considerations appropriate to the MISP processor as described in detail in the sections below. Other conventional operating system functions can be implemented in the usual way.

The operating system handles two types of queues and a table in low memory that are relevant to process scheduling and swapping. Each of the queues has appropriate pointers set up, which are maintained by operating system routines. The manipulation of these pointers is a critical section of code, and only one process at a time is allowed access to these critical sections. Critical sections are protected by the use of semaphores, which are described in the next section. The Process Ready Queue (PRQ) contains the Process Status Word (PS) and the Program Counter (PC) of processes ready and waiting to be run. If more than one priority of ready processes is desired, more than one Process Ready Queue could be defined and maintained in software. The Process Wait Queue (PWQ) contains the Process Status Word (PS) and Program Counter (PC) of processes that have been swapped out of the pipeline while they are waiting for some external event to occur. The Active Process Table (APT) contains the Process Identification number (PID) of those processes currently running in the pipeline.

3.1.2. Process Control

Several instructions were added to the instruction set or modified for the MISP environment to allow process control and swapping functions to occur. These instructions are SPL, CPSW, RPSW, TSET, and CTST. These instructions are included in a table in Chapter 1 and are described here in greater detail.

CPSW can be executed only in Kernel Mode. It is used to change the Process Status Word of a stream when a process swap is occurring. This instruction replaces the Process I. D. (PID) of a process, and initializes the Trace bit and the condition codes. It also replaces the Priority bit.

SPL is used to set the Priority Mode lower. SPL changes the stream from Kernel Mode to User Mode. The process is in Kernel Mode while performing process swapping functions and updating the queues. It can change itself to User Mode when these operations are finished.

RPSW is an instruction that reads the value of the stream's Process Status Word. RPSW is useful to maintain the Process Ready Queue, the Process Wait Queue, and the Active Process Table. User processes are also allowed to execute RPSW.

There are some sections of code which are critical sections. Critical sections must not be executed by more than one process at a time. The exclusivity of critical sections is protected by semaphore registers. Semaphores can also be used to gain access to hardware resources shared by more than one process. The semaphore registers are actually special hardwired locations which are addressed by specified memory addresses (i.e., memory-mapped). These registers can be modified by the Kernel Mode instructions TSET and CTST. When TSET is executed, a semaphore register is tested and set if it is zero. If it is not zero, the semaphore register is read repeatedly until it becomes zero when another process releases it. When a process has set the semaphore to one, that process can then execute those critical sections of code or access those hardware resources that are protected by that specific semaphore. CTST is used to clear the semaphore register when the process is through with the section of code or the hardware

resource and is willing to release it for use by other processes. Only Kernel Mode processes can execute TSET and CTST. These semaphore registers are protected by the memory controller's protection mechanism and cannot be accessed by User Mode processes. This feature allows the processes to interact in a highly concurrent multiprogramming environment.

Process swapping is used when a process is finished, when a process is being halted, when the processor is being bootstrapped, or when special traps are performed. Process swaps are initiated by one of the trap instructions. Each type of trap instruction traps to a particular operating system routine through a particular pair of reserved low memory locations. The trap instructions automatically put the stream that will execute the operating system routine in Kernel Mode. The Process Status Word (PS) and the Program Counter (PC) of the process that was running in that stream of the pipeline are pushed on the process stack. The operating system routine replaces the process that was running, and executes in the same stream that had been running that process. The detailed mechanism of the trap instructions is described in the next section. If the new process is to be taken from the Process Ready Queue (or Process Wait Queue), the operating system routines perform the process swap by using TSET to gain access to the Process Ready Queue (or Process Wait Queue). The operating system can then remove the new Process Status Word (PS) value and the new Program Counter (PC) value from the front of the queue and store them in registers. The pointers then can be updated to point to the next process in the queue. The CTST instruction can then free up the semaphore that protects the sections of code which manipulate the relevant pointers. If the old process is to be put on the Process Wait Queue, a similar procedure using semaphores is followed to update the Process Wait Queue. The Active Process Table can also be updated to indicate the process that is about to be started. The new value of the Process Status Word (PS) is then put in the stream's PS using the CPSW. This action will change the Process I. D. (PID) of the stream as well as the Priority (P) bit. The condition codes are also initialized. The process can then execute a JMP instruction and jump to the instruction pointed to by the new value of the Program Counter, which was stored in a register. The operating system routine is thereby replaced

in the stream by this new process.

RESET is an instruction which can be used to bootstrap the system or restart the system. Only Kernel Mode processes can execute the RESET instruction. This instruction clears out the ERR and FULL bits in MBUF and also clears out any memory requests that were put in the MRB and the MWB. A clock is started, whose value is put into the Stream I. D. of the streams as they enter Segment One. For eight clock periods, the START trap opcode will be loaded into each stream's Instruction Register during Segment One. The mechanism for the START trap opcode is described in the next section, but it results in a trap to an operating system routine. This operating system routine will start a new process in the pipeline and replace itself with this new process.

3.1.3. Traps and Interrupts

Traps and interrupts are used to execute routines that are not in the normal instruction stream of a specific process. System traps provide access to trace routines, illegal instruction routines, memory error routines, and process termination routines (for halting a process or initializing the system when the processor is reset). There are also user traps which can be used to provide for calls to I/O monitors, debugging packages, and user-defined interpreters, and that can terminate the processes when they are finished and initiate other processes by swapping them into the pipeline. When a trap occurs, the state of the process is saved by pushing the Process Status Word (PS) and the Program Counter (PC) of the process that is currently executing. A new PS and PC are then loaded into the stream's registers from two-word trap vectors which contain a new value for the Process Status Word and the starting address of the special trap routine. The sequence of actions, which is listed below, is almost identical for all traps. The exceptions are described in the following text.

(1) The trap instruction is loaded into the IR from IR CODES in Segment One.

(2) A dummy read is executed to check for memory errors.

(3) The PS of the current process is pushed onto the process stack.

(4) The PS for the oper ing system trap routine is loaded from the trap vector.

(5) The process is set to Kernel Mode.

(6) The PC of the current process is pushed onto the process stack.

(7) The PC for the operating system trap routine is loaded from the trap vector.

(8) The operating system trap routine is executed.

The trap instructions are assigned permanent fixed addresses in low memory for their trap vectors. The next instruction fetch after a trap will be from the routine designated by the address in the trap vector. Table 3.1 shows the trap instructions along with the address of their associated trap vectors. These addresses are slightly different from those in the *PDP-11* architecture. The address listed in the table is the first word of the two-word trap vector where the new Program Counter value is located. However, the first fetch in the register transfer language implementation will be from the second word, where the new Process Status Word is located. The address of the location that contains the new value of the Process Status Word is found in the lower eight bits of the trap and interrupt instructions. After the new Process Status Word is fetched, the Instruction Register is decremented by two to point to the starting address of the trap routine. The stream is set to Kernel Mode during trap operations, since operating system queues may need to be updated. After the routine is executed, the return to the original process is done via a RTI or a RTT instruction at the end of the routine. Since RTI and RTT restore the Process Status Word, including its Priority bit, RTI and RTT are executable only by Kernel Mode processes. To assure system security, the Process Status Word should be checked by the operating system routine when the trap first occurs, and compared to the Process Status Word which is restored to the process. These Process Status Words should be identical. This protection is required since the Process

Table 3.1 Addresses of Trap Vectors

Trap or instruction	Description	Memory Address (hex)
	User Trap Class	
EMT	Terminate process	0018
TRAP	User-defined trap	001C
IOT	User-defined I/O trap	0010
BPT	Breakpoint trap	000C
	System Trap Class	
Halted Process	Halted process instruction	0018
Trace trap	Traced instruction trap	000C
Illegal Instr.	Reserved instruction trap	0008
Memory Error	Memory management violations or ECC failure	0004
START	Start processes	0000
INTR	Interrupted process instruction	0020

Status Word is generally saved on a process stack located in an unprotected area of memory.

There are four instructions in the User Trap class. These are EMT, TRAP, IOT, and BPT. These instructions are executed in the normal instruction stream of a User Mode or Kernel Mode process. The sequence of actions for these instructions is the same as those listed above, except that no forced instruction load from IRCODES and no dummy read are necessary. Any pending write requests are cleared out of the Memory Write Buffer (MWB) by the read request generated by reading the instruction before the instruction is executed, so all memory errors are cleared and no dummy read is necessary.

When a process is finished, its instruction stream is ended by an EMT instruction. The typical trap execution is performed, and an operating system routine executes in the same stream that had been running the process that finished. This routine swaps processes in the manner described above, gaining access to the Process Ready Queue through setting the semaphore, removing the new Process Status Word (PS) value and the new Program Counter (PC) value from the front of the queue, storing them in registers, and releasing the semaphore. The registers can also be initialized in this routine, if necessary. Then the Active Process Table is updated, the CPSW instruction is used to change the Process Status Word, and the JMP instruction is executed to jump to the starting address of the new process. In this way, the operating system routine effectively replaces itself with a new process. TRAP, IOT, and BPT instructions execute traps in the same manner as described above, but the operating system routines perform functions appropriate for the particular trap instruction being executed. These functions are described in the PDP-11 Handbook [DEC81].

There are six instructions in the System Trap Class. These are the Halted Process instruction, the Trace trap, the Illegal Instruction trap, the Memory Error trap, the START instruction, and the Interrupted process instruction (INTR). The execution of each of these traps is very similar. They all follow the pattern described above for traps, with a few minor differences. The opcodes for these instructions are located in IR CODES, which is a buffer in Segment One containing the hard-wired opcodes. When its specific conditions are satisfied in Segment One, one of these opcodes is loaded into the

Instruction Register. The particular sequence of microinstructions for that particular instruction is then followed. The particular sequence for each type of trap is described below.

A process can be halted when a Kernel Mode process (halting process) uses a HLTP instruction to put another stream's Process I. D. (PID) in the Halt Buffer. When this PID is loaded in, the HLTP instruction simultaneously sets the HT flag. The HT flag tells the stream whether the process it tried to halt was active in the pipeline. When each process arrives at Segment One, it compares its PID to the one in the Halt Buffer. If there is a match, it sets the STOP bit in its Dynamic Status Register (STAT) and executes the rest of its current instruction. The halted process also clears the HT flag as soon as a PID match occurs to indicate that the process to be halted was found. When the halting process again arrives in Segment One, one cycle later, it loads the HT flag into the V bit in its PS. The V bit can be checked with a conditional branch instruction. It also clears out the Halt Buffer and the HT flag. When the halted process arrives in Segment One after completing its instruction, the STOP bit is already set. This causes the Halted Process Instruction opcode from the IR CODES to be loaded. The usual trap procedure is then followed. It executes a dummy read from the address in its Program Counter to guarantee that there are no outstanding memory requests or memory management errors meant for this stream. If there are no errors, the old Process Status Word and Program Counter are pushed, and the starting address of the system trap routine is loaded into the Program Counter. This routine is the same one that is executed when EMT is executed, and results in a new process being swapped in to replace the system trap routine.

A Trace trap occurs at the end of an instruction when the T bit of the Process Status Word is set and the instruction can be traced. The RTT and TSET instructions cannot be traced. When RTT is executed, the trace is suppressed, so that RTT can be used to return from a Trace trap. For a process to be traced, the Dynamic Status Word (STAT) of the process must be zero, which means that no wait cycles are occurring and the process is not about to execute a system trap. The process also cannot be preparing to service an interrupt. In Segment Five, when these conditions are met, the TRACE bit of STAT is set.

On the process' next cycle, the TRACE bit is checked in Segment One, and if it is set, the Instruction Register will be loaded with the Trace trap instruction from the IR CODES buffer. The usual trap procedure is then followed. A dummy read will be executed to clear out any memory requests for this stream, and the Process Status Word and Program Counter will be pushed as usual. The operating system trap routine is the same as that executed when the BPT trap is executed. This routine can record pertinent information and status information, and then return to the original process with a RTT instruction.

In Segment Two, the instructions are checked to see if they are legal. If the opcodes are not legal, if the priority is not valid for a particular instruction, or if an addressing mode is not legal, the IIT bit in STAT is set to one. The process executes NOP's in Segments 3-8. When the process arrives in Segment One on its next cycle, the Illegal Instruction opcode is loaded into the Instruction Register from IR CODES. The normal trap procedure is then executed. A dummy read is performed, and the Process Status Word and the Program Counter are pushed as usual. The PS and PC are loaded for an operating system routine from the trap vector. An operating system routine is executed which outputs an appropriate error message and starts another process by performing the process swap procedure mentioned in the previous section.

Whenever a memory read occurs, an error bit is sent from the memory controller with the returned data. This bit is put in ERR[sid], which is a bit in the Memory Buffer (MBUF). Write error indications, if any, are attached by the memory controller chip to the following read data return for that process. Therefore, dummy read transactions are required before certain traps are executed in order to clear out all pending write requests for that process and check for write memory errors. In the PLA for Segment One, the ERR bit associated with the instruction stream being decoded is checked. If this bit is set, a memory error trap occurs. Instead of loading the memory data from the Memory Buffer, the Instruction Register is loaded with the opcode for a memory error trap. No dummy read is done in this trap, since a memory error has already been detected. The Process Status Word and the Program

Counter are pushed as usual, and the PS and PC for an operating system routine that handles memory errors are loaded. This operating system routine can poll the memory controller to determine the type of error and output appropriate error messages. If necessary, diagnostic tests can be performed on the memory. Once the error is identified, the process can either be terminated or restarted, depending on the type of error.

The START trap is used to start the processor as well as to reset it after it has been running. The processor can be reset by pulsing the RESET pin or by executing the RESET instruction in any stream. Only Kernel Mode processes can execute the RESET instruction. This instruction clears out the ERR bits in MBUF, and clears out any memory requests that were put in the MRB or MWB. A 3-bit clock counter is started, whose value is put into the Stream I. D. of the streams as they enter Segment One. For eight clock periods, the START trap opcode is loaded into each stream's Instruction Register during Segment One. The START trap does not push the Process Status Word and Program Counter as usual, but instead loads a new Program Counter and a new Process Status Word from the trap vector in memory. The new Program Counter points to the operating system routine that starts a new process from the queue and initializes the general purpose registers. For a cold start, preliminary instructions can bootstrap the system on the first execution of this routine by reading in system code and establishing initial queues from some input device.

Interrupts are handled very much like traps are handled. INT is the interrupt line that comes into the processor. This line can be set by an interrupting device, or by an interrupt control chip. This line can set the INTLINE flip-flop. INTLINE is the line that is checked by the PLA in Segment Five. To be interrupted, the following conditions must be satisfied:

(1) The process must be in User Mode.

(2) The process must be on the last cycle of an instruction.

(3) The process must have no pending memory accesses.

(4) The process must not be executing a System Trap instruction.

(5) The process must not be about to be halted.

With eight streams, there should always be a sufficient number of streams with processes in this state so that interrupts can be handled in a timely fashion. When the process is interrupted, the following actions then occur:

(1) The interrupt instruction is loaded into the IR from IR CODES.

(2) A dummy read is executed to check for memory errors.

(3) The PS of the interrupted process is pushed onto the process stack.

(4) The PS of the interrupting handling process is loaded from location 0022 hex.

(5) The process is set to Kernel Mode.

(6) The PC of the interrupted process is pushed onto the process stack.

(7) The starting address of the interrupt handling process is loaded from location 0020 hex.

Location 20 may contain the address of the interrupt handling process for a specific device, or it may contain the address of a general interrupt handling process which will poll the devices or interrogate a memory mapped vector interrupt location in the interrupt controller chip to determine which device needs to be serviced. Location 22 contains a Process Status Word for the interrupt handling process which will be loaded into the PS of the stream. The priority of the process will be set to Kernel Mode automatically. Once the interrupting device is known, the interrupt handling process can load in a Process I.D. to execute a specific interrupt service routine. For short interrupts, the interrupt is simply serviced, and then the stream executes a RTI or RTT instruction, which causes a return to the interrupted process. If longer interrupt service routines are required, the interrupt handling process can gain access to the Process Ready Queue using an appropriate semaphore, and then put a Program Counter and a Pro-

cess Status Word of an interrupt service routine on the Process Ready Queue. The semaphore could then be released by the CTST instruction, and the interrupt service routine would execute as soon as one of the other processes terminated and was removed from the pipeline. Alternatively, the interrupted process could be removed to the Process Ready Queue and the interrupt service routine could be executed at once.

The RTI instruction effectively restores a process that has been interrupted. When an RTI instruction is executed, the saved Program Counter value is popped off the top of the stack and loaded into the PC. The PS is restored by popping the value off the stack as well. These are the values that were saved from the process that was interrupted. By restoring the Program Counter, the process will resume execution, starting with its next instruction. The RTT instruction does the same thing as the RTI instruction, except that RTT inhibits a trace trap, while RTI permits a trace trap. Since both RTI and RTT restore the Process Status Word, they are executable only by Kernel Mode processes.

3.2. MISP System Configuration

An example of a complete system configuration for the MISP processor is shown in Figure 3.1. Although other system configurations are possible, a description of this configuration follows. This system consists of the MISP processor along with a memory controller, an interrupt controller and a bus controller.

The memory controller handles memory requests for the processor. A full function memory controller is described here. It is a very sophisticated memory management unit that can provide extended memory addressing with virtual memory capabilities for each process. Memory protection and access violation handling are also implemented. Each stream has its own page tables that map addresses and define the limits of its memory. It has page descriptor registers to limit the access to certain pages. These functions are possible since the processor sends the Stream I.D.'s to the memory controller which

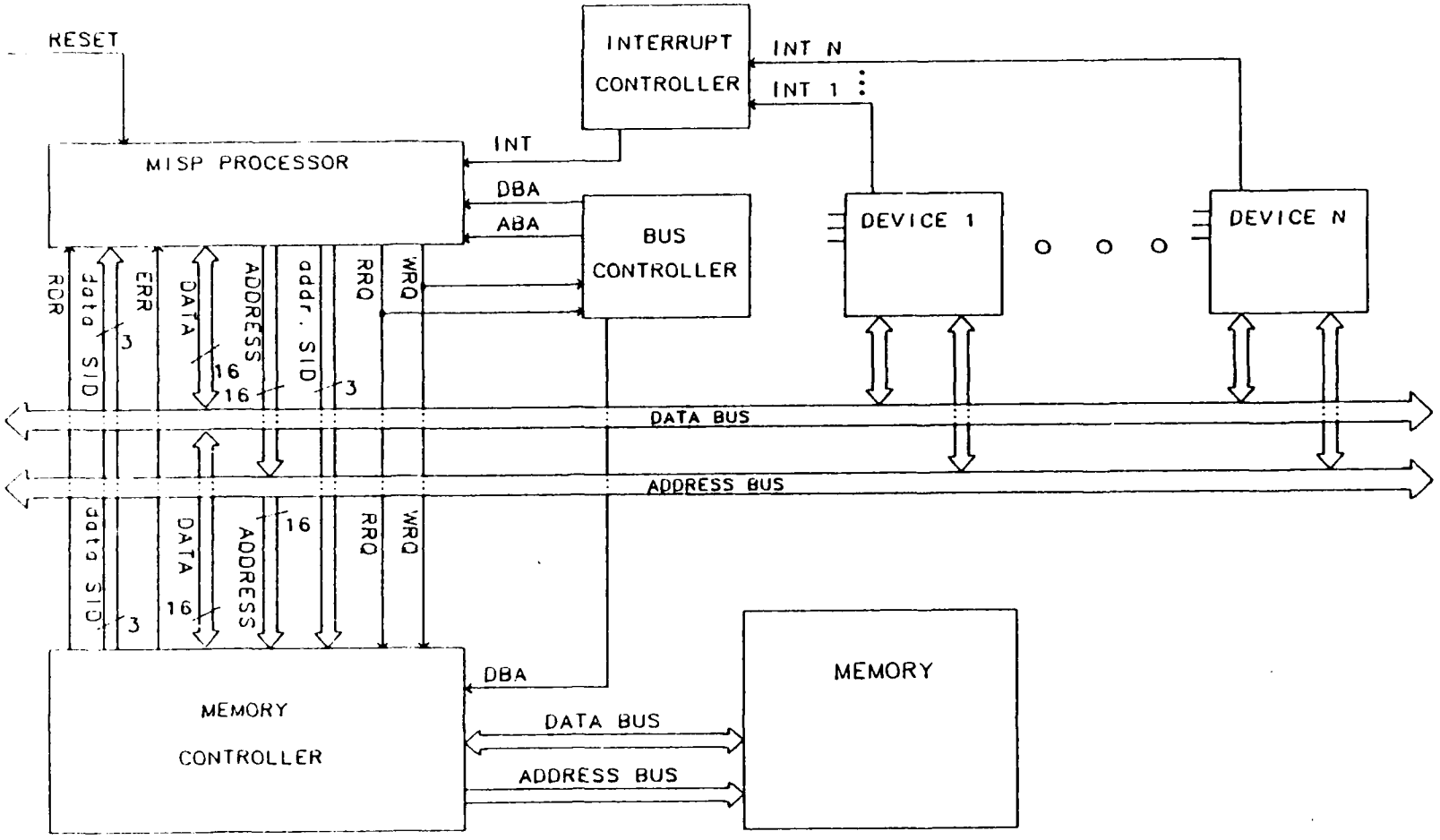


Figure 3.1 MISP System Configuration

can access or retain a copy of the Active Process Table. For a read request, it receives the 16-bit address along with a three-bit SID associated with the address, and the Read Request (RRQ) pin, which tells the memory controller that a read request is occurring. For a write, the memory controller receives the 16-bit address along with the three-bit SID, and sixteen bits of data to be written. It also receives the Write Request (WRQ) pin, which tells it that a write request is occurring. It is assumed that the memory controller can overlap the read request with the return of read data from prior read requests. When the memory controller gains access to the data bus, it sends the 16-bit data word along with an ERR bit. It also sends a three-bit SID associated with the data which will identify which stream the data belongs to. This SID must be separate from the address SID so that read requests can be overlapped with the return of the read data. The memory controller signals this return with the Read Data Return (RDR) line.

The bus controller controls access to both the address bus and the data bus. One possible implementation of this bus controller is shown and this example is described below. MISP uses memory-mapped I/O, so data is transferred to devices using the Data Bus and the Address Bus. The bus controller handles requests for the bus from many devices. MISP sends the signals Read Request (RRQ) and Write Request (WRQ) when it needs to access memory or a device. (In the diagram, the corresponding bus request signals for the other devices were omitted for clarity.) When a device makes a bus request, the bus controller decides which device gets priority, and grants bus access to a device by sending the appropriate bus available signals. On the MISP processor itself, these signals are Data Bus Available (DBA) and Address Bus Available (ABA). A logically distinct DBA is sent to the memory controller for read data return. (The corresponding bus available signals for other devices were omitted.) Another possible implementation for the bus control is to have a fixed priority daisy chain system, with MISP being the lowest priority device. This is cheaper to implement, but it requires that there be relatively low bus traffic among the other devices so that MISP would not be slowed down while waiting for bus access.

The interrupt controller shown receives the interrupt request lines from several devices. It then pulses the INT signal for one cycle, which signals the processor that interrupt service is requested. The interrupt controller can handle many different priority levels of devices, and can determine which device needs to be serviced. The interrupt controller is memory-mapped, so that when MISP services the interrupt, MISP can determine which device needs to be serviced by reading a vector from the interrupt controller that identifies the highest priority interrupting device and/or the initial address of the appropriate interrupt service routine. A cheaper method to implement the interrupt control is to have a daisy chain priority system which generates the INT signal. The devices would then have to be polled to determine which device is interrupting. In some implementations, the interrupt controller and the bus controller would need a connection to the system buses for memory mapped locations, direct memory access, and/or I/O channel implementations.

This system provides full system capability, while maintaining a minimum amount of hardware on the MISP processor chip itself. The MISP processor is therefore suitable for use in a broad range of system configurations.

4. CONCLUSION

4.1. Implementation

The MISP processor is very suitable for VLSI implementation. It is suggested that CMOS technology should be used for the layout of the chip, since CMOS is known to consume much less power. Low power is extremely important for VLSI systems as circuit densities are increased. Another advantage for CMOS is that circuit design is easier. Static ratioless circuits are possible in CMOS, which are easier to design than NMOS ratioed logic. This section gives an estimate of the circuit complexity of the MISP processor using CMOS technology. The complexity and size of each of the main functional blocks can be estimated by looking at the number of inputs, outputs, and an estimated transistor gate count. Either a PMOS gate or an NMOS gate is counted as one gate (e.g., one inverter is two gates). Table 4.1 lists these three quantities for the main functional blocks of the MISP processor. This table shows that the total processor is 36240 gates, which is quite reasonable for VLSI implementation.

There are some possible improvements to the MISP architecture which could be areas for future research. Table 4.1 shows that the number of gates in the PLA's is 35% of the total number of transistors. Although these PLA's have a higher density of transistors than random logic, the control circuitry would still be a large percentage of the total chip area. We believe it might be possible to decrease the area of the control logic by changing the instruction set. Recently, LOAD/STORE architectures with simplified instruction sets, [FIT81], [PTR82], [HEN82], [RAD83], have become quite popular. A variety of LOAD/STORE architectures has been designed so as to reduce the complexity of the instruction set, which results in reduced control logic, reduced machine cycle times, and reduced design time for the microprocessor. In the RISC architecture [FIT81], [PTR82], the Berkeley group was able to reduce the control section of the chip to about 4% of the total transistors by using a reduced instruction set. Although this figure includes the effect of a 32-bit data path and a very large number of registers, while the MISP has only a 16-bit data path and 7 (including the SP) general purpose registers per process, the control logic of MISP would still be decreased by using a simpler instruction set. The number

Table 4.1 I/O and Transistor Count for Main Functional Blocks

Hardware Block	Inputs	Outputs	Gates
Register File	26	32	6390
MBUF	27	19	1440
Segment 1	163	136	1510
Segment 2	155	136	1250
Segment 3	131	137	2600
Segment 4	120	129	1010
MRB	22	21	1380
Segment 5	129	156	2210
MWB	38	37	2460
Segment 6	121	166	1080
Segment 7	115	149	1000
Segment 8	117	130	1160
PLA1	23	25	1950
PLA2	20	12	2900
PLA3	23	20	1880
PLA4	19	5	800
PLA5	26	12	2640
PLA6	18	4	740
PLA7	19	4	980
PLA8	20	6	860
Total # PLA gates			12750
Total # gates for MISP			36240

of bits of the Instruction Register used as PLA inputs could be decreased to 5 or 6 by decreasing the number of addressing modes, and using addressing modes other than register direct for only a few instructions. The effect of a reduced instruction set could be explored in future MISP research. Other techniques for reducing the control area include predecoding, wherein some of the status bits which are used as PLA inputs are combined in simple combinational logic whose outputs are used as inputs to the PLA. It might also be possible to implement a master predecode PLA, which can generate inputs for other PLA's, thereby reducing the number of inputs for the other PLA's. These signals for other PLA's would have to be tagged with the proper SID and then stored until the stream reaches the appropriate segment, so the technique may not be area-efficient. It is nevertheless worth investigating.

One of the major design challenges for VLSI technology is to make the designs fault-tolerant. Since this design is only a prototype for evaluation of the MISP architecture, we simplified the design by not including extra circuitry to provide fault-tolerance capabilities. However, we believe that concurrent error detection techniques should be incorporated into future implementations of the MISP architecture. Some techniques which could be used for future designs are mentioned below.

Traditional stuck-at fault models are often inadequate for modeling faults in VLSI technology. A more appropriate model for VLSI is a model presented in [PAT82], which assumes that physical failures are confined to a small area on the chip or a cluster of components. Patel and Fung present a method called RESO, which recomputes shifted operands and detects errors in a bit-sliced ALU satisfying their broad fault model. This method could perhaps be applied to the ALU in MISP, although the segment design would have to be modified since the method requires two computations for every ALU operation. The ALU could, however, be spread over two segments. Since the time required for the computation in each segment would then be approximately half, the recomputation could occur in the second half of the cycle. Some additional logic would be required so that the comparisons and error recovery logic could be overlapped with other operations. It is found in [BAN82] and [FUC84] that most faults in PLA's are unidirectional, and a method using a modified Berger code is presented in

[MAK83]. This technique is then improved in [FUC84], and it could be applied to make the PLA's on the MISP processor fault-tolerant. Concurrent error detection is incorporated into the design of a microprogram control unit in [WON83]. Several techniques are integrated into the design, including Berger codes, totally self-checking equality checkers, and a scheme to make the stack fault-tolerant using "register tags." Many of these techniques could also be implemented in a future MISP architecture to provide concurrent error detection for the control signals. A parity scheme or a Berger code could be used for the register file. Although the complexity and additional area required for such circuitry is considered inappropriate for our current prototype design goal, concurrent error detection should be incorporated into a future implementation.

The MISP processor offers a performance advantage over conventional microprocessor architectures. Because of its pipelined architecture with eight segments, we expect it to have a speed-up of nearly eight. This processor has an operating system capable of permitting interactions between processes. A mechanism for handling traps and interrupts was added to enable MISP to fit into a complete system. These capabilities enable the processor to be used in a real multitasking environment.

In the MISP processor, pin utilization is increased over a single processor. All eight processes use the same ports and buses to make memory requests. Both read requests and write requests use the same address pins, so it is expected that the pins will be busy most of the time. The data pins for the read data return are the same as the data pins for the write request. This sharing yields very high pin utilization. In addition, hardware resources are well-utilized. The streams share a dual read port and a single write port for the register file, and only the registers in the register file need to be replicated. Since it is a pipelined architecture, pipeline resources can be shared (although some added logic is required). These features minimize the area of the processor.

The MISP architecture is also very cost-effective. While most multiprocessor architectures require several chips, the MISP multiprocessor can fit on a single chip. Since only 50 pins are required, it is a relatively cheap package compared to most multiprocessor architectures. Several characteristics,

primarily high pin bandwidth, high memory utilization, and high resource utilization, make it a cost-effective alternative to conventional multiprocessor architectures, which should be seriously considered for VLSI implementation.

APPENDIX A. MISP INSTRUCTION SET

The complete MISP instruction set is given below. Much of the description is taken from Appendix A of [ARC82]. The User Trap, the System Trap, the Loop, and the Process Control classes of instructions have been modified. Some of the instructions have been clarified by describing the execution of the operation. The description of the execution of the operation follows the symbol "> >."

Single Operand Instructions (SOP)

CLR – Destination is set to zero.

COM – Destination is logically complemented.

INC – Destination is incremented.

DEC – Destination is decremented.

NEG – Destination is replaced by binary radix complement of original value.

TST – Set condition flags N and Z according to value of destination.

ASR – Arithmetically shift destination right.

ASL – Arithmetically shift destination left.

ROR – Rotationally shift destination right.

ROL – Rotationally shift destination left.

SWAB–Swap bytes of destination.

ADC – Add carry flag to destination.

SBC – Subtract carry flag from destination.

SXT – Destination gets zero if N flag clear, else destination gets negative one.

Branch Instructions (BRAN)

BR – Unconditional branch, PC gets PC plus two times offset.

BNE – Branch if Z = 0.

BEQ – Branch if Z = 1.

BPL – Branch if N = 0.

BMI – Branch if N = 1.

BVC – Branch if V = 0.

BVS – Branch if V = 1.

BCC – Branch if C = 0.

BCS – Branch if C = 1.

BGE – Branch if XOR(N,V) = 0.

BLT – Branch if XOR(N,V) = 1.

BGT – Branch if (Z OR XOR(N,V)) = 0.

BLE – Branch if (Z or XOR(N,V)) = 1.

BHI – Branch if C = 0 and Z = 0.

BLOS – Branch if (C or Z) = 1.

BHIS – Branch if C = 0.

BLO – Branch if C = 1.

Loop (LOOP) Instruction

SOB – Subtract one from register and branch if not equal to 0.
Used for looping.
>> subtract one from register,
if register is not equal to zero,
subtract twice offset from PC.

Condition Code Operators (COND)

CLC – Clear C.

SEC – Set C.

CLV – Clear V.

SEV – Set V.

CLZ – Clear Z.

SEZ – Set Z.

CLN – Clear N.

SEN – Set N.

CCC – Clear all condition codes.

SCC – Set all condition codes.

User Trap (UTRAP) Class

EMT – Emulator trap.
Used to end processes. Also loaded
into IR when a process is halted by another process.
>> push old PS,
load PS from location 001A hex,
set priority high (if not already),
push old PC,
load PC from location 0018 hex,
results in jump to location loaded in address 0018 hex.

TRAP–User-defined trap.
Used for arbitrary user traps.
>> push old PS,
load PS from location 001E hex,
set priority high (if not already),
push old PC,
load PC from location 001C hex,
results in jump to location loaded in address 001C hex.

BPT – Breakpoint trap.
Used for trace trap.
>> push old PS,
load PS from location 000E hex,
set priority high (if not already),
push old PC,
load PC from location 000C hex,
results in jump to location loaded in address 000C hex.

IOT – Input/output trap.
Used for user trap routine
>> push old PS,
load PS from location 0012 hex,
set priority high (if not already),
push old PC,
load PC from location 0010 hex,
results in jump to location loaded in address 0010 hex.

Return From Interrupt (RETI) Instructions

RTI – Return from interrupt or operating system TRAP routine.
(Kernel Mode only)
Will not inhibit a trace trap. If the T bit is set,
trace trap occurs immediately after the RTI instruction.
>> pop return address from top of stack into PC,
pop old PS value from stack into Process Status Word.

RTT – Same as RTI, except RTT inhibits trace trap.
(Kernel Mode only)
If the T bit is set, the pending "T" trap
will execute after the next instruction.
>> pop return address from top of stack into PC,
pop old PS value from stack into Process Status Word.

Jump Instructions Class (JUMP)

JMP – Jump to location.
>> PC gets destination.

JSR – Jump to subroutine.
>> push current linkage register,
move old PC to new linkage register,
set PC to destination address

RTS – Return from subroutine.
>> move value in current linkage register to PC,
pop old linkage register value into current linkage reg.,
return to address in PC.

Double Operand (DOP) Instructions

MOV –Destination gets source.

ADD – Destination gets source plus destination.

SUB – Destination gets destination minus source.

BIT – Adjust N, V, and Z flags according
to AND(source, destination).

BIC – Destination gets AND(COMP(source), destination)

BIS – Destination gets OR(source, destination).

XOR –Destination gets XOR(source, destination).

CMP – Compares the source and destination operands
and sets the condition codes.
The compare is source minus destination.

Process Control (PCNTL) Class

RESET–Resets processor.
(Kernel Mode only)
> > RESETLINE will be set.
the CT registers of all processes will be set to 0,
all processes will finish the cycle with NOP's,
a START instruction will be loaded into each stream's IR.

SPL – Set priority lower.
Used before an exit from the OS scheduling routine
when the new process needs to be low priority.
> > sets bit in PS for User Mode

HLTP–Causes another process to halt.
(Kernel Mode only)
> > reads a given register which contains the PID
of the process which is to be halted,
loads this value into the Halt Buffer (HLTB);
the process will be halted if it is in the pipe;
the overflow bit is cleared if the process was found.

CPSW–Change Process Status Word to new word.
(Kernel Mode only)
Used for swapping processes.
> > reads a given register which contains the new PID,
loads this value into PID in PS.

RPSW–Read Process Status Word
Used to read a PSW to determine status information
and to find out its Process I.D. after interrupt.
> > reads PID from PS and loads value into register.

TSET –Test and set semaphore location.
(Kernel Mode only)
> > fetches the address of the semaphore from the destination.
tests and sets the semaphore if it's not already set,
waits in loop until the bit becomes cleared.

CTST– Clears semaphore location.
(Kernel Mode only)
> > fetches the address of the semaphore from the destination.
clears the semaphore location.

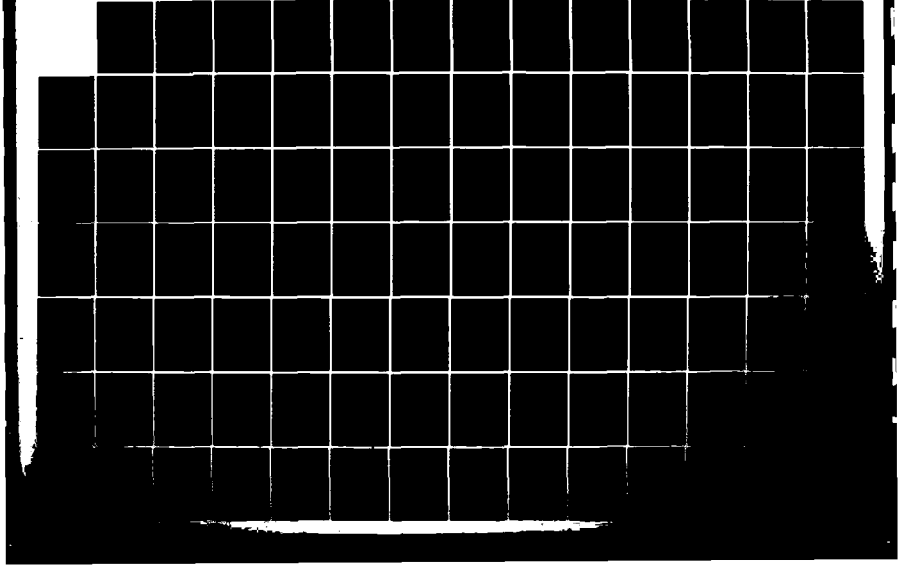
System Trap (STRAP) Class

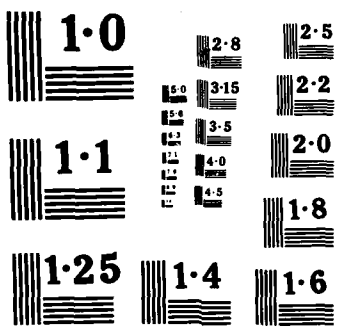
INTR –Interrupted process instruction.
> > in Segment 5, when conditions are met,
set INTERRUPT, and clear INTLINE,
in Segment 1, load the interrupt
opcode into the Instruction Register,
push old PS,
load PS from location 0022 hex,
set priority high (if not already),
push old PC,
load PC from location 0020 hex,
results in trap to an operating system routine.

Halted Process–Halted process instruction.
> > in Segment 1, when conditions are met,
set STOP, and clear HT,
finish that instruction,
starting on the next cycle, load the
halted process opcode into the Instruction Register,
push old PS,
load PS from location 001A hex,
set priority high (if not already),
push old PC,
load PC from location 0018 hex,
results in trap to an operating system routine.

AD-A156 314 DESIGN FOR MISP: A MULTIPLE INSTRUCTION STREAM SHARED PIPELINE PROCESSOR(U) ILLINOIS UNIV AT URBANA COMPUTER SYSTEMS GROUP L M PEDERSEN DEC 84 CSG-37 2/3

UNCLASSIFIED N00039-80-C-0556 F/G 9/2 NL





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

Trace Trap –Traced instruction trap
>>in Segment 5, when conditions are met, set TRACE,
in Segment 1, load the trace trap opcode
into the Instruction Register,
push old PS,
load PS from location 000E hex,
set priority high (if not already),
push old PC,
load PC from location 000C hex,
results in trap to an operating system routine.

Illegal Instruction –Reserved instruction trap
>>when illegal instruction detected, set IIT,
NOP rest of that cycle,
in Segment 1, load the illegal instruction
trap opcode into the Instruction Register,
push old PS,
load PS from location 000A hex,
set priority high (if not already),
push old PC,
load PC from location 0008 hex,
results in trap to an operating system routine.

Memory Error –Memory management violations
>>in Segment 1, when memory error is detected,
load memory error trap opcode into
into the Instruction Register,
push old PS,
load PS from location 0006 hex,
set priority high (if not already),
push old PC,
load PC from location 0004 hex,
results in trap to an operating system routine.

START–Start processes or reset processor
>>in Segment 1, when RESETLINE is set,
load START opcode into the Instruction Register,
push old PS,
load PS from location 0002 hex,
set priority high (if not already),
push old PC,
load PC from location 0000 hex,
results in trap to an operating system routine.

APPENDIX B. SIGNALS, REGISTERS AND BUFFERS

Abbrev.	Name	Description
ABA	Address Bus Available	Line which comes from an off-chip bus controller. Indicates that the Address Bus is available to the processor.
ALU	Arithmetic Logic Unit	Performs the arithmetic and logic operations for the processor. Located in Segment Three.
APT	Active Process Table	Software table located in protected memory that is updated by an operating system routine. Contains the Process I. D. (PID) of the processes currently running in the pipeline.
C	Carry flag	The Carry flag is located in the Process Status Word (PS). Indicates a carry condition exists when set to one.
CC	Condition Codes	The Condition Codes are located in the Process Status Word (PS). They include the Negative (N), Zero (Z), Overflow (V), and Carry (C) flags.
CCBLOCK	Condition Code Block	The CCBLOCK is located in Segment Four. It is a block of logic that execute the Condition Code Operators. This block sets or clears the bits that are masked off in the opcode.
COUNTER	Counter	The Counter is located in Segment One. When RESETLINE is first set, the counter is started. Four eight cycles, the value of the clock will be put in the CT registers of the streams as they enter Segment One. After eight cycles, RESETLINE will be reset.
DBA	Data Bus Available	Line which comes from an off-chip bus controller. Indicates that the Data Bus is available to the processor.

Abbrev.	Name	Description
DW	Data Wait flag	The Data Wait flag is located in the Dynamic Status Word (STAT). Indicates that the stream is waiting for data to return, and is executing wait states until the data arrives.
ENRB	Enable Read Buffer line	This line is generated by logic in Segment Five. When a microinstruction is attempted that deposits a read request in the Memory Read Buffer (MRB) and the MRB is not full, ENRB is high, which gates the memory request into the buffer.
ENWB	Enable Write Buffer line	This line is generated by logic in Segment Six. When a microinstruction is attempted that deposits a write request in the Memory Write Buffer (MWB) and the MWB is not full, ENWB is high, which gates the memory request into the buffer.
ERR[sid]	Error flag	Located in the MBUF. Each stream has its own error flag addressed by its Stream I. D. (SID). It is set when the data from memory returns with an error.
FULL[sid]	Full bit	Located in the MBUF. Each stream has its own Full bit. Indicates that the data has returned from memory, and that the buffer location currently contains the data.
HBF	HLTB Full wait flag	Located in Dynamic Status Buffer (STAT). Indicates that the Halt Buffer (HLTB) is full, and the process is waiting for it to become empty.
HBFULL	Halt Buffer Full flag	Associated with the HLTB. Indicates that the Halt Buffer is currently in use.
HLTB	Halt Buffer	Buffer located in Segment One. Used for halting another process. The PID of the process to be halted is put in the buffer. Each process compares its PID to the one in the Halt Buffer.
HT	HLTB status flag	Flag associated with the Halt Buffer (HLTB). Set when a Process I. D. is put in the HLTB. Cleared when the process to be halted is found in the pipeline.

Abbrev.	Name	Description
IIT	Illegal Instruction Trap flag	Located in the Dynamic Status Word (STAT). Indicates that an illegal instruction was detected, and an Illegal Instruction Trap is being executed.
IR	Instruction Register	A sixteen-bit register carried through the pipeline with the stream. Contains the opcode of the current instruction being executed.
INT	Interrupt pin	This is the pin which is pulsed when an interrupt occurs. It can either be pulsed by an interrupt controller or a device, depending on how the system is set up. When it is pulsed, it causes the line INTLINE to be set.
INTERRUPT	Interrupt status flag	Located in the Dynamic Status Register (STAT). Indicates that a stream is being interrupted.
INTLINE	Interrupt flag	Indicates that an interrupt is pending. INTLINE is a line set when a pulse comes to the interrupt pin. Cleared by hardware in Segment Five.
MBUF	Memory Data Buffer	Located in Segment One. Stores the data that has been read from memory. Contains eight locations of 16 bits plus an ERR bit and an EMPTY bit associated with each location. Locations are addressed by the Stream Identification number (SID). MBUF[sid] refers to the location for the stream "sid."
MEMERR	Memory Error status flag	Located in the Dynamic Status Register (STAT). Indicates that a memory error trap is occurring.
MRB	Memory Read Buffer	Located in Segment Five. FIFO buffer containing up to eight locations for memory read requests. Each location contains a three-bit SID and a sixteen-bit address.
MWB	Memory Write Buffer	Located in Segment Six. FIFO buffer containing eight locations for memory write requests. Each location contains a three-bit SID, a sixteen-bit address, and a sixteen-bit data word.

Abbrev.	Name	Description
N	Negative flag	The Negative flag is located in the Process Status Word (PS). Indicates a negative condition exists when set to one.
PRQ	Process Ready Queue	A queue located in protected memory, which is maintained by an operating system routine that is protected by a semaphore register. This queue contains the Process Status Word (PS) and Program Counter (PC) of processes ready and waiting to be run.
PWQ	Process Wait Queue	A queue located in protected memory, which is maintained by an operating system routine that is protected by a semaphore register. This queue contains the Process Status Word (PS) and Program Counter (PC) of processes that have been swapped out of the pipeline while they are waiting for some external event to occur.
PC	Program Counter	A sixteen-bit register carried through the pipeline with the stream. Contains the address of the next instruction.
PID	Process I. D.	Located in the eight higher-order bits of the Process Status Word (PS). Identifies the process that is currently running in the stream.
PS	Process Status Word	A sixteen-bit register carried through the pipeline with the stream. Contains the PID, PRIORITY, and five flags--T, N, Z , V, and C. Contains the current of the process.
PRIORITY	Priority flag	Located in the Process Status Word. Indicates that the stream is currently in Kernel Mode when set. The stream is in User Mode when the bit is clear.
RDR	Read Data Return	Line which comes from off-chip and goes to the Memory Data Buffer. Indicates that the data is being returned from the bus. Gates the data into the buffer.

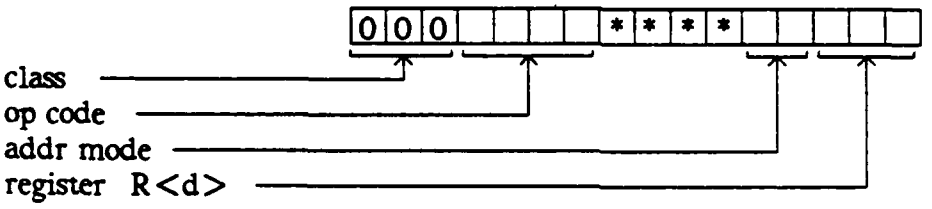
Abbrev.	Name	Description
RESETLINE	Reset line	Line which is set by the RESET pin from off-chip or can be set by the instruction RESET. When set, this line causes the CT register for all processes to be cleared to 0, and it clears out the full bits in the buffers. It causes the START instruction to be loaded into each stream's IR in Segment One. The line is reset after eight clocks.
RRQ	Read Request line	This line is generated in Segment Five when there is a read request ready to go out and no write requests in the MWB. The line is sent off-chip to a bus controller or to the memory controller.
RRW	Read Request Wait flag	The Read Request Wait flag is located in the Dynamic Status Word (STAT). It indicates that the Memory Read Buffer (MRB) is full, and that the stream is waiting to make a read request. The stream will execute wait states until it can deposit its read request in the MRB.
SENDR	Send Read	SENDR is generated from logic in Segment Six and sent to the MRB in Segment Five. It is high when the address bus is available, and a read request has been made. It gates the address out on to the bus from the Memory Read Buffer to initiate a read request.
SENDW	Send Write	SENDR is generated in Segment Six and is sent to the MWB. It is high when both the address and the data buses are available and a write request was made. It gates the address and the data out on to the bus from the Memory Write Buffer to initiate a write request.
STOP	Stop status bit	Located in the Dynamic Status Register (STAT). Indicates that a stream has identified its PID in the Halt Buffer, and that it will be halted after its present instruction.

Abbrev.	Name	Description
T	Trace trap bit	Located in the Process Status Word (PS). Indicates that a trace trap should occur after every possible instruction is executed. Only certain instructions such as RTT and some System Trap instructions can not be traced.
TRACE	Trace Status flag	Located in the Dynamic Status Register (STAT). Indicates that the Trace Trap is in the process of executing.
V	Overflow flag	The Overflow flag is located in the Process Status Word (PS). Indicates that an overflow condition exists when set to one.
WRQ	Write Request line	This line is generated in Segment Six when there is a write request ready to go out. The line is sent off-chip to a bus controller or to the memory controller.
WRW	Write Request Wait flag	The Write Request Wait flag is located in the Dynamic Status Word (STAT). It indicates that the Memory Write Buffer (MWB) is full, and that the stream is waiting to make a write request. The stream will execute wait states until it can deposit its write request in the MWB.
SID	Stream I. D.	The Stream I. D. is a three-bit register that is carried through the pipeline with a stream. Each stream has a different SID. It identifies which stream is in a given segment, and is used to tag memory requests and data returned from memory. The Stream Identification numbers are initialized with a RESET instruction.
Z	Zero flag	The Zero flag is located in the Process Status Word (PS). It indicates that a zero condition exists when set to one.

APPENDIX C. INSTRUCTION EXECUTION CYCLES BY INSTRUCTION CLASS

The register transfer language for the instructions is given below. The Instruction Register format is given first, then the microcode for each of the four addressing modes is given. The code (register transfer language) given for each instruction assumes that there are no prevailing error conditions. Error conditions and special execution sequences follow the normal instruction set. These sequences include the Interrupt Instruction, the Trace Trap, the Halted Process Instruction, the Memory Error Trap, and Illegal Instruction Trap. Status bits are checked at certain times in the execution of instructions, and if those conditions are met, the special execution sequence occurs. It is assumed throughout the microcode that the required checks are being done. One example of this is that whenever data is requested from memory, if the data is not returned yet, the stream will just execute a cycle of NOPs until the data appears. Other status bits that must be checked are the Read Request Wait flag (RRW), the Write Request Wait flag (WRW), and the Halt Buffer Full flag (HBF). When these microinstructions are executed, if the buffers are full, the stream will have to execute a cycle of NOPs until the buffer becomes available. These checks are assumed, but were left out of the microcode for ease of readability.

Single Operand Instructions (SOP) format



Instructions included =

CLR, COM, INC, DEC, NEG, TST, ASR, ASL, ROR, ROL, SWAB,ADC, SBC, SXT

RTL description of instruction type

mode 0 a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<d>, CT := 1; /*Temp. gets operand.

MD1 := <op> MD1; /*Perform operation.

CC := <op> CC; /*Flag update.

MRB := PC, PC := PC + 2; /*Request next instruction.

R<d> := MD1; /*Register gets result.

CT := CT - 1. /*Set counter.

mode 1 a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<d>, CT := 2; /*Temp. 1 gets operand address.

MRB := MD1; /*Request operand.

CT := CT - 1, MD3 := MD1; /*Temp. 3 has operand address.

b. MD2 := MBUF[sid]; /*Temp. 2 has operand

MD1 := <op> MD2; /*Perform operation.

CC := <op> CC; /*Flag update.

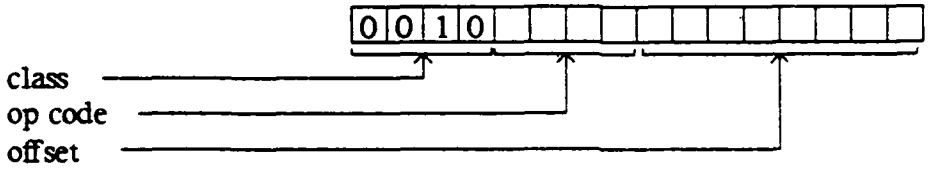
MRB := PC, PC := PC + 2; /*Request instruction.

MWB := (MD3,MD1); /*Put result in location.

CT := CT - 1. /*Decrement counter.

```
mode 2   a. IR := MBUF[sid]; /*Load instruction.
           MD1 := R<d>, CT := 3; /*Temp. 1 gets operand address.
           MRB := MD1; /*Request operand.
           CT := CT - 1, MD3 := MD1; /*Temp. 3 has operand address.
         b. MD2 := MBUF[sid]; /*Temp. 2 has operand
           MD1 := <op> MD2; /*Perform operation.
           CC := <op> CC; /*Flag update.
           MWB := (MD3,MD1); /*Store result in memory.
           CT := CT - 1, MD1 := MD3; /*Decrement counter.
         c. MD1 := MD1 + 2; /*Increment register value.
           MRB := PC, PC := PC + 2; /*Request instruction.
           R<d> := MD1; /*Update Register value.
           CT := CT - 1. /*Decrement counter.

mode 3   a. IR := MBUF[sid]; /*Load instruction.
           MD1 := R<d>, CT := 3; /*Temp. 1 gets base address.
           MRB := PC, PC := PC + 2; /*Request offset.
           CT := CT - 1. /*Set counter.
         b. MD2 := MBUF[sid]; /*Temp. 2 gets offset.
           MD1 := MD1 + MD2; /*Compute address of operand.
           MRB := MD1; /*Request operand.
           CT := CT - 1, MD3 := MD1; /*Temp. 3 gets operand address.
         c. MD2 := MBUF[sid]; /*Temp. 2 gets operand.
           MD1 := <op> MD2; /*Perform operation.
           CC := <op> CC; /*Flag update.
           MRB := PC, PC := PC + 2; /*Request instruction.
           MWB := (MD3, MD1); /*Put result in location.
           CT := CT - 1. /*Decrement counter.
```

Branch Instructions (BRAN) format



Instructions =

BR, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO

RTL description

all modes

a. IR := MBUF[sid]; /*Load instruction.

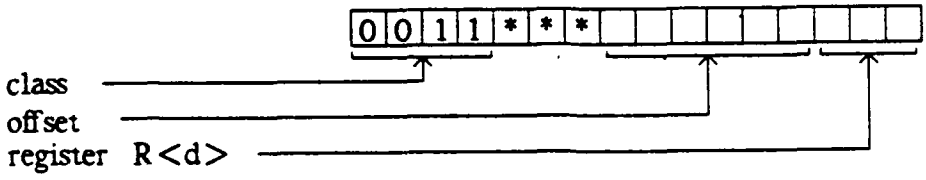
CT := 1; /*Set counter.

IF condition THEN PC := PC + 2* IR[7:0]; /*Compute branch address.

MRB := PC, PC := PC + 2; /*Request instruction at branch address.

CT := CT - 1. /*Decrement counter.

Loop Instruction (LOOP) format



Instruction = SOB

RTL description

all modes

a. IR := MBUF[sid]; /*Load instruction.

MD2 := R<d>. CT := 2; /*Temp. 2 gets register value.

MD2 := MD2 - 1; /*Subtract one from register.

Z := TEST MD2; /*Test for zero.

R<d> := MD2; /*Update register value.

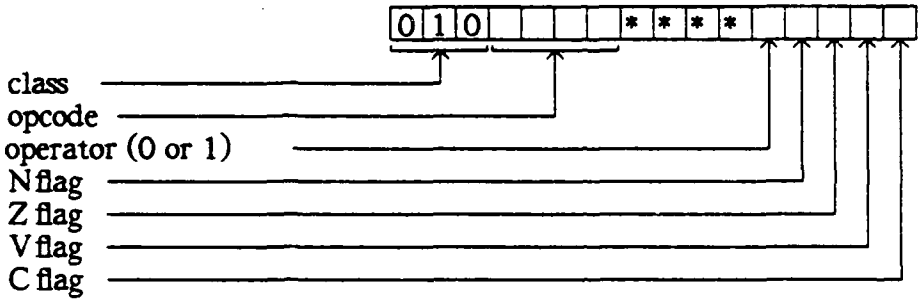
CT := CT - 1; /*Decrement counter.

b. IF Z = 0 THEN PC := PC - 2* IR[8:3]; /*Compute loop address.

MRB := PC. PC := PC - 2; /*Request instruction at loop address.

CT := CT - 1. /*Decrement counter.

Condition Code Operators (COND) format



Instructions = CLC, CLV, CLN, CLZ, CCC, SEC, SEV, SEZ, SEN, SCC

RTL description

a. IR := MBUF[sid]; /*Load instruction.

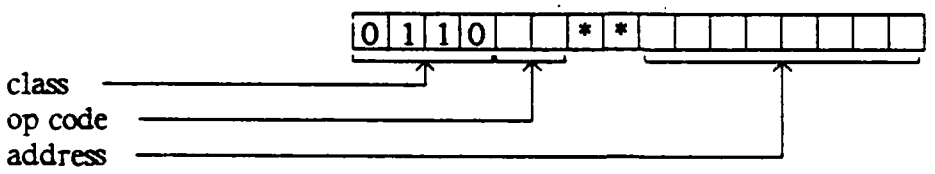
CT := 1; /*Set counter.

PS[code] := IR[4]; /*Set masked flags in PSW.

MRB := PC, PC := PC - 2; /*Request instruction.

CT := CT - 1. /*Set counter.

User Trap (UTRAP) Class
Trap Instructions format



Instructions = EMT, TRAP, IOT, BPT

RTL description

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R6, CT := 3; /*Temp. 1 is SP.

MD1 := MD1 - 2; /*Decrement SP.

MRB := IR[7:0], IR := IR - 2; /*Request PS from trap vector.

MWB := (MD1, PS); /*Push PS.

R6 := MD1; /*Update SP.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets new PS.

MD1 := MD1 - 2; /*Decrement SP.

PS := MD2, PRIORITY := 1; /*Load in new PS.

MRB := IR[7:0]; /*Request new PC.

MWB := (MD1, PC); /*Push old PC.

R6 := MD1; /* Update SP.

CT := CT - 1; /*Decrement counter.

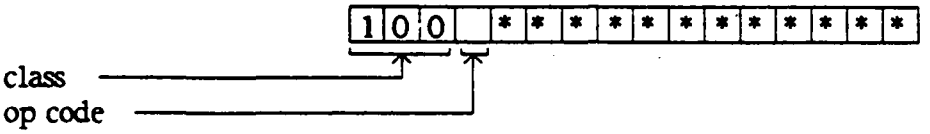
c. MD2 := MBUF[sid]; /*Temp. 2 gets new PC.

PC := MD2; /*Load in new PC.

MRB := PC, PC := PC + 2; /*Request next instruction.

CT := CT - 1. /*Decrement counter.

Return from Interrupt (RETI) format



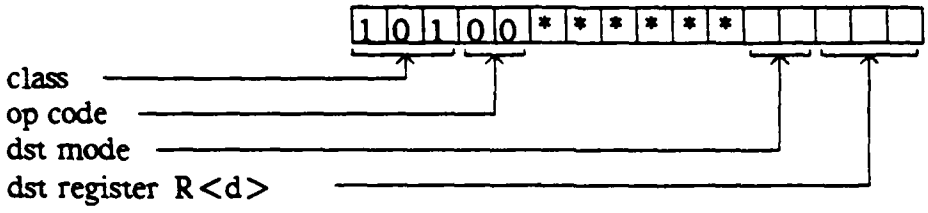
Instructions = RTI, RTT

RTL description

```
        a. IR := MBUF[sid]; /*Load instruction.
IF PRIORITY := 0  THEN IIT := 1.  GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
  {         MD1 := R6. CT := 3;  /*Temp. 1 gets SP.
            MRB := MD1, MD1 := MD1 + 2;  /*Request for pop PC off stack.
            R6 := MD1;  /*Update SP.
            CT := CT - 1;  /*Decrement counter.
          b. MD2 := MBUF[sid];  /*Temp. 2 gets return address from stack.
            PC := MD2;  /*The return address is loaded into PC.
            MRB := MD1, MD1 := MD1 + 2;  /*Request for pop of PS off stack.
            R6 := MD1;  /*Update SP.
            CT := CT - 1;  /*Decrement counter.
          c. MD2 := MBUF[sid];  /*Temp. 2 gets PS from stack.
            PS := MD2;  /*PS for process loaded in.
            MRB := PC, PC := PC + 2;  /*Request next instruction.
            CT := CT - 1.  /*Decrement counter.
  }
```

Jump Instructions Class (JUMP)

Jump Instruction format



Instruction = JMP

RTL description

mode 0 Not Allowed. Results in illegal instruction trap.

a. IR := MBUF[sid]; /*Load instruction.

IIT := 1; /*Illegal instruction trap flag set.

mode 1 a.IR := MBUF[sid]; /*Load instruction.

MD1 := R<d>, CT := 1; /*Load jump address.

PC := MD1; /*Load jump address in PC.

MRB := PC, PC := PC + 2; /*Request next instruction.

CT := CT - 1. /*Decrement counter.

mode 2 a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<d>, CT := 2; /*Load jump address.

PC := MD1; /*Load jump address in PC.

CT := CT - 1; /*Decrement counter.

b. MD1 := MD1 + 2; /*Increment register value.

MRB := PC, PC := PC + 2; /*Request next instruction.

R<d> := MD1; /*Update register value.

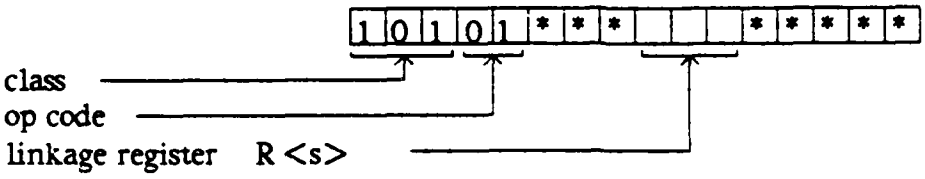
CT := CT - 1; /*Decrement counter.

mode 3 a. IR := MBUF[sid]; /*Load instruction.

```
MD1 := R<d>, CT := 3;  /*Temp. 1 is base address.
MRB := PC, PC := PC + 2;  /*Request offset.
CT := CT - 1;  /*Decrement counter.
b. MD2 := MBUF[sid];  /*Temp. 2 is offset.
MD1 := MD1 + MD2;  /*Compute jump address.
CT := CT - 1;  /*Decrement counter.
c. PC := MD1;  /*PC set for jump address.
MRB := PC, PC := PC + 2;  /*Request next instruction.
CT := CT - 1.  /*Decrement counter.
```

Jump Instructions Class (JUMP)

RTS instruction format



Instruction = RTS

RTL description

a. IR := MBUF[sid]; /*Load instruction.

MD2 := R<s>, MD1 := R6, CT := 2;

/*Temp. 2 gets return address. Temp. 2 is SP.

PC := MD2; /*PC gets return address.

MRB := MD1, MD1 := MD1 - 2; /*Request pop of old linkage.

R6 := MD1; /*Update stack pointer.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Pop old linkage into Temp. 2.

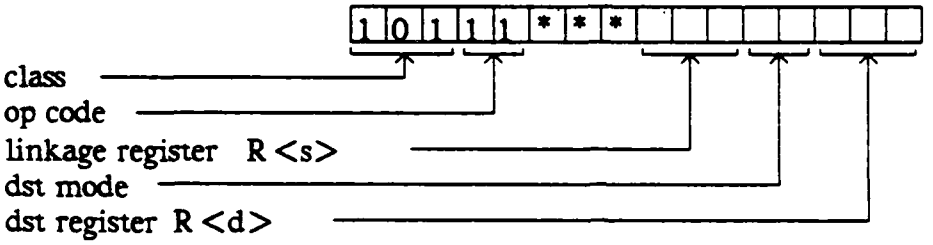
MRB := PC, PC := PC - 2; /*Request next instruction.

R<s> := MD2; /*Write old linkage into linkage register.

CT := CT - 1. /*Decrement counter.

Jump Instructions Class (JUMP)

JSR instruction format



Instruction = JSR

RTL description

mode 0 Not Allowed. Results in illegal instruction trap.

a. IR := MBUF[sid]; /*Load instruction.

IIT := 1; /*Illegal instruction trap flag set.

RTL description

mode 1 a. IR := MBUF[sid]; /*Load instruction.

MD2 := R<s>, MD1 := R6, CT := 2; /*MD2 has old linkage reg. MD1 is SP.

MD1 := MD1 - 2; /*Decrement SP

MWB := (MD1, MD2); /*Push old linkage reg.

R6 := MD1; /*Update SP.

CT := CT - 1, MD2 := PC; /*MD2 gets return address.

b. MD1 := R<d>; /*MD1 gets subroutine address.

PC := MD1; /*PC gets subroutine address.

MRB := PC, PC := PC + 2; /*Request instruction.

R<s> := MD2; /*Linkage reg. gets return address.

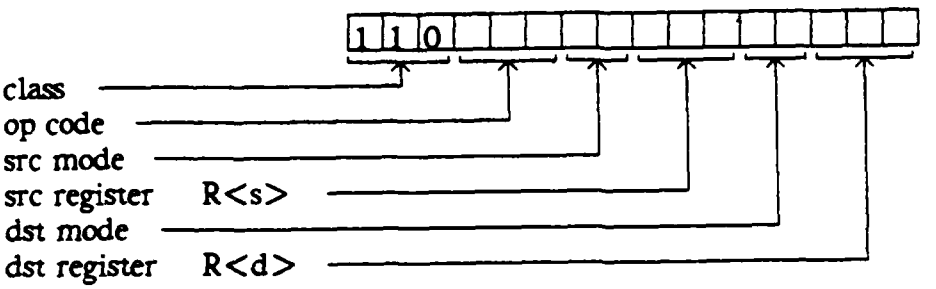
CT := CT - 1. /*Decrement counter.

mode 2 a. IR := MBUF[sid]; /*Load instruction.

MD2 := R<s>, MD1 := R6, CT := 3; /*MD2 has old linkage reg., MD1 is SP.

```
        MD1 := MD1 - 2; /*Decrement SP.
        MWB := (MD1, MD2); /*Push old linkage reg.
        R6 := MD1; /*Update SP.
        CT := CT - 1, MD2 := PC; /*MD2 gets return address.
      b. MD1 := R<d>; /*MD1 has subroutine address.
        PC := MD1; /*PC gets subroutine address.
        R<s> := MD2; /*Linkage reg. gets return address.
        CT := CT - 1; /*Decrement counter.
      c. MD1 := MD1 + 2; /*Autoincrement register value.
        MRB := PC, PC := PC + 2; /*Request subroutine instruction.
        R<d> := MD1; /*Register is autoincremented.
        CT := CT - 1. /*Decrement counter.

mode 3  a. IR := MBUF[sid]; /*Load instruction.
        MD2 := R<s>, MD1 := R6, CT := 3; /*MD2 gets old linkage, MD1 gets SP.
        MD1 := MD1 - 2; /*Decrement SP.
        MRB := PC, PC := PC + 2; /*Request offset.
        MWB := (MD1,MD2); /*Push old linkage.
        R6 := MD1; /*Update SP.
        CT := CT - 1; /*Decrement counter.
      b. MD2 := MBUF[sid]; /*MD2 is offset.
        MD1 := R<d>; /*MD1 gets base address.
        MD1 := MD1 + MD2; /*Calculate subroutine address.
        CT := CT - 1, MD2 := PC; /*MD2 is return address.
      c. PC := MD1; /*Load subroutine address.
        MRB := PC, PC := PC + 2; /*Request subroutine instruction.
        R<s> := MD2; /*Load linkage register with return address.
        CT := CT - 1; /*Decrement counter.
```

Double Operand Instructions (DOP) format



Instructions =

MOV(B), ADD, SUB, BIT(B), BIC(B), BIS(B), XOR, ASH

RTL description

mode 0,0

a. IR := MBUF[sid]; /*Load instruction.

MD2 := R<s>, MD1 := R<d>, CT := 1; /*Temp. registers get operands.

MD1 := MD2 <op> MD1; /*Perform operation.

CC := <op> CC; /*Flag update.

MRB := PC, PC := PC + 2; /*Request next instruction.

R<d> := MD1; /*Write result to register.

CT := CT - 1; /*Decrement counter.

mode 0,1

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<d>, CT := 2; /*Temp. 1 gets address of destination.

MRB := MD1; /*Request destination operand.

CT := CT - 1, MD3 := MD1; /*Temp. 3 get destination address.

b. MD2 := MBUF[sid]; /*Temp. 2 gets destination operand.

MD1 := R<s>; /*Temp. 1 gets source operand.

MD1 := MD1 <op> MD2; /*Perform operation.

CC := <op> CC; /*Update flags.

```
        MRB := PC, PC := PC + 2;  /*Request next instruction.
        MWB := (MD3, MD1);  /*Write results.
        CT := CT - 1. /*Decrement counter.

mode 0,2

    a. IR := MBUF[sid];  /*Load instruction.
        MD1 := R<d>, CT := 3;  /*Temp. 1 gets address of destination.
        MRB := MD1;  /*Request destination operand.
        CT := CT - 1, MD3 := MD1;  /*Temp. 3 get destination address.
    b. MD2 := MBUF[sid];  /*Temp. 2 gets destination operand.
        MD1 := R<s>;  /*Temp. 1 gets source operand.
        MD1 := MD1 <op> MD2;  /*Perform operation.
        CC := <op> CC;  /*Update flags.
        MWB := (MD3,MD1);  /*Write results.
        CT := CT - 1, MD1 := MD3;  /*Temp. 1 gets address.
    c. MD1 := MD1 + 2;  /*Increment register value.
        MRB := PC, PC := PC + 2;  /*Request next instruction.
        R<d> := MD1;  /*Increment destination register.
        CT := CT - 1. /*Decrement counter.

mode 0,3

    a. IR := MBUF[sid];  /*Load instruction.
        MD1 := R<d>, CT := 3;  /*Temp. 1 is destination base address.
        MRB := PC, PC := PC + 2;  /*Request destination offset.
        CT := CT - 1;  /*Decrement counter.
    b. MD2 := MBUF[sid];  /*Temp. 2 is destination offset.
        MD1 := MD1 + MD2;  /*Calculate address of destination operand.
        MRB := MD1;  /*Request destination operand.
        CT := CT - 1, MD3 := MD1;  /*Temp. 3 get address of destination.
    c. MD2 := MBUF[sid];  /*Temp. 2 gets destination operand.
        MD1 := R<s>;  /*Temp. gets source operand.
```

MD1 := MD1 <op> MD2; /*Perform operation.

CC := <op> CC; /*Update flags.

MRB := PC, PC := PC + 2; /*Request next instruction.

MWB := (MD3, MD1); /*Write results.

CT := CT - 1. /*Decrement counter.

mode 1,0

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<s>, CT := 2; /*Temp. 1 gets source address.

MRB := MD1; /*Request source operand.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets source operand.

MD1 := R<d>; /*Temp. 1 gets destination operand.

MD1 := MD2 <op> MD1; /*Perform operation.

CC := <op> CC; /*Update flags.

MRB := PC, PC := PC + 2; /*Request next instruction.

R<d> := MD2; /*Write results.

CT := CT - 1. /*Decrement counter.

mode 1,1

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<s>, CT := 3; /*Temp. 1 gets address of source operand.

MRB := MD1; /*Request source operand.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets source operand.

MD1 := R<d>; /*Temp. 1 gets address of destination operand.

MRB := MD1; /*Request destination operand.

CT := CT - 1, MD3 := MD1, MD1 := MD2;

/*Temp. 3 gets address of destination, Temp. 1 gets source operand.

c. MD2 := MBUF[sid]; /*Temp. 2 gets destination operand.

MD1 := MD1 <op> MD2; /*Perform operation.

CC := <op> CC; /*Update flags.

MRB := PC, PC := PC + 2; /*Request next instruction.

MWB := (MD3, MD1); /*Write results.

CT := CT - 1. /*Decrement counter.

mode 1,2

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<s>, CT := 4; /*Temp. 1 gets address of source operand.

MRB := MD1; /*Request source operand.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets source operand.

MD1 := R<d>; /*Temp. 1 gets address of destination operand.

MRB := MD1; /*Request destination operand.

CT := CT - 1. MD3 := MD1, MD1 := MD2;

/*Temp. 3 gets address of destination. Temp. 1 gets source operand.

c. MD2 := MBUF[sid]; /*Temp. 2 gets destination operand.

MD1 := MD1 <op> MD2; /*Perform operation.

CC := <op> CC; /*Update flags.

MWB := (MD3, MD1); /*Write results.

CT := CT - 1. MD1 := MD3; /*Temp. 1 gets register value.

d. MD1 := MD1 + 2; /*Increment register value.

MRB := PC, PC := PC + 2; /*Request next instruction.

R<d> := MD1; /*Increment register.

CT := CT - 1; /*Decrement counter.

mode 1,3

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<d>, CT := 4; /*Temp. 1 gets destination base address.

MRB := PC, PC := PC + 2; /*Request destination offset address.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets destination offset address.

```
        MD1 := MD1 + MD2;  /*Calculate address of destination operand.
        MRB := MD1;  /*Request destination operand.
        MD3 := MD1, CT := CT - 1;  /*Temp. 3 gets address of destination.
      c. MD2 := MBUF[sid];  /*Temp. 2 gets destination operand.
        MD1 := R<s>;  /*Temp. 2 gets address of source operand.
        MRB := MD1;  /*Request source operand.
        CT := CT - 1, MD1 := MD2;  /*Temp. 1 gets destination operand.
      d. MD2 := MBUF[sid];  /*Temp. 2 gets source operand.
        MD1 := MD2 <op> MD1;  /*Perform operation.
        CC := <op> CC;  /*Update flags.
        MRB := PC, PC := PC + 2;  /*Request next instruction.
        MWB := (MD3, MD1);  /*Write results.
        CT := CT - 1.  /*Decrement counter.

mode 2,0

      a. IR := MBUF[sid];  /*Load instruction.
        MD1 := R<s>, CT := 2;  /*Temp. 1 gets source address.
        MRB := MD1, MD1 := MD1 + 2;  /*Request source and autoincrement.
        R<s> := MD1;  /*Autoincrement source register.
        CT := CT - 1;  /*Decrement counter.
      b. MD2 := MBUF[sid];  /*Temp. 2 gets source operand.
        MD1 := R<d>;  /*Temp. 1 gets destination operand.
        MD1 := MD2 <op> MD1;  /*Perform operation.
        CC := <op> CC;  /*Update flags.
        MRB := PC, PC := PC + 2;  /*Request next instruction.
        R<d> := MD2;  /*Write results.
        CT := CT - 1;  /*Decrement counter.

mode 2,1

      a. IR := MBUF[sid];  /*Load instruction.
        MD1 := R<d>, CT := 3;  /*Temp. 1 gets address of destination.
```

```
        MRB := MD1;  /*Request destination operand.
        CT := CT - 1, MD3 := MD1;  /*Temp. 3 saves destination address.
    b. MD2 := MBUF[sid];  /*Temp. 2 gets destination operand.
        MD1 := R<s>;  /*Temp. 1 gets address of source operand.
        MRB := MD1, MD1 := MD1 + 2;  /*Request source and autoincrement.
        R<s> := MD1;  /*Autoincrement source register.
        CT := CT - 1;  /*Decrement counter.
    c. MD2 := MBUF[sid];  /*Temp. 2 gets source operand.
        MD1 := MD2 <op> MD1;  /*Perform operation.
        CC := <op> CC;  /*Update flags.
        MRB := PC, PC := PC + 2;  /*Request next instruction.
        MWB := (MD3, MD1);  /*Write results.
        CT := CT - 1.  /*Decrement counter.

mode 2,2

    a. IR := MBUF[sid];  /*Load instruction.
        MD1 := R<s>, CT := 4;  /*Temp. 1 gets address of source operand.
        MRB := MD1, MD1 := MD1 + 2;  /*Request source and autoincrement.
        R<s> := MD1;  /*Autoincrement source register.
        CT := CT - 1;  /*Decrement counter.
    b. MD2 := MBUF[sid];  /*Temp. 2 gets source operand.
        MD1 := R<d>;  /*Temp. 1 gets address of destination operand.
        MRB := MD1;  /*Request destination operand.
        CT := CT - 1, MD3 := MD1, MD1 := MD2;
        /*Temp. 3 gets address of destination. Temp. 1 gets source operand.
    c. MD2 := MBUF[sid];  /*Temp. 2 gets destination operand.
        MD1 := MD1 <op> MD2;  /*Perform operation.
        CC := <op> CC;  /*Update flags.
        MWB := (MD3, MD1);  /*Write results.
        CT := CT - 1, MD1 := MD3.  /*Temp. 1 gets register value.
    d. MD1 := MD1 + 2;  /*Increment register value.
```

MRB := PC, PC := PC - 2; /*Request next instruction.

R<d> := MD1; /*Increment register.

CT := CT - 1; /*Decrement counter.

mode 2,3

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<d>, CT := 4; /*Temp. 1 gets destination base address.

MRB := PC, PC := PC + 2; /*Request destination offset address.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets destination offset address.

MD1 := MD1 + MD2; /*Calculate address of destination operand.

MRB := MD1; /*Request destination operand.

MD3 := MD1, CT := CT - 1; /*Temp. 3 gets address of destination.

c. MD2 := MBUF[sid]; /*Temp. 2 gets destination operand.

MD1 := R<s>; /*Temp. 2 gets address of source operand.

MRB := MD1, MD1 := MD1 + 2; /*Request source and autoincrement.

R<s> := MD1; /*Autoincrement source register.

CT := CT - 1, MD1 := MD2; /*Temp. 1 gets destination operand.

d. MD2 := MBUF[sid]; /*Temp. 2 gets source operand.

MD1 := MD2 <op> MD1; /*Perform operation.

CC := <op> CC; /*Update flags.

MRB := PC, PC := PC - 2; /*Request next instruction.

MWB := (MD3, MD1); /*Write results.

CT := CT - 1. /*Decrement counter.

mode 3,0

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<s>, CT := 3; /*Temp. 1 gets base address for source.

MRB := PC, PC := PC + 2; /*Request offset for source.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets source offset.

MD1 := MD1 + MD2; /*Calculate address of source operand.

MRB := MD1; /*Request source operand.

CT := CT - 1; /*Decrement counter.

c. MD2 := MBUF[sid]; /*Temp. 2 gets source operand.

MD1 := R<d>; /*Temp. 1 gets destination operand.

MD1 := MD2 <op> MD1; /*Perform operation.

CC := <op> CC; /*Update flags.

MRB := PC, PC := PC + 2; /*Request next instruction.

R<d> := MD1; /*Write results to register.

CT := CT - 1. /*Decrement counter.

mode 3,1

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<s>, CT := 4; /*Temp. 1 gets base address for source.

MRB := PC, PC := PC + 2; /*Request offset for source.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets offset for source.

MD1 := MD1 + MD2; /*Calculate address of source operand.

MRB := MD1; /*Request source operand.

CT := CT - 1; /*Decrement counter.

c. MD2 := MBUF[sid]; /*Temp. 2 gets source operand.

MD1 := R<d>; /*Temp. 1 gets address of destination.

MRB := MD1; /*Request destination operand.

CT := CT - 1, MD3 := MD1, MD1 := MD2;

/*Temp. 3 gets address of destination, Temp. 1 gets source operand.

d. MD2 := MBUF[sid]; /*Temp. 2 gets destination operand.

MD1 := MD1 <op> MD2; /*Perform operation.

CC := <op> CC; /*Update flags.

MRB := PC, PC := PC + 2; /*Request next instruction.

MWB := (MD3, MD1); /*Write results.

CT := CT - 1. /*Decrement counter.

mode 3,2

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<s>, CT := 5; /*Temp. 1 gets base address for source.

MRB := PC, PC := PC + 2; /*Request offset for source.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets offset for source.

MD1 := MD1 + MD2; /*Calculate address of source operand.

MRB := MD1; /*Request source operand.

CT := CT - 1; /*Decrement counter.

c. MD2 := MBUF[sid]; /*Temp. 2 gets source operand.

MD1 := R<d>; /*Temp. 1 gets address of destination.

MRB := MD1; /*Request destination operand.

CT := CT - 1, MD3 := MD1, MD1 := MD2;

/*Temp. 3 gets address of destination, Temp. 1 gets source operand.

d. MD2 := MBUF[sid]; /*Temp. 2 gets destination operand.

MD1 := MD1 <op> MD2; /*Perform operation.

CC := <op> CC; /*Update flags.

MWB := (MD3, MD1); /*Write results.

CT := CT - 1, MD1 := MD3; /*Temp. 1 gets register contents.

e. MD1 := MD1 + 2; /*Increment destination register value.

MRB := PC, PC := PC + 2; /*Request next instruction.

R<d> := MD1; /*Increment destination register.

CT := CT - 1; /*Decrement counter.

mode 3,3

a. IR := MBUF[sid]; /*Load instruction.

MD1 := R<s>, CT := 5; /*Temp. 1 gets source base address.

MRB := PC, PC := PC + 2; /*Request source offset.

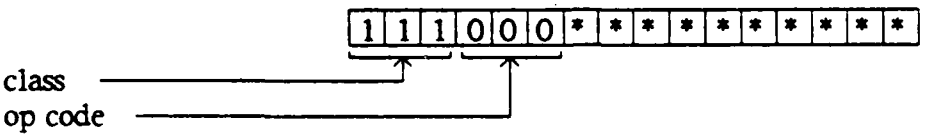
CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 gets source offset.

MD1 := MD1 + MD2; /*Calculate address of source operand.

```
MRB := MD1;  /*Request source operand.
CT := CT - 1;  /*Decrement counter.
c. MD2 := MBUF[sid];  /*Temp. 2 gets source operand.
MD1 := R<d>;  /*Temp. 1 gets destination base address.
MRB := PC, PC := PC + 2;  /*Request destination offset address.
CT := CT - 1, MD3 := MD2;  /*Temp. 3 gets source operand.
d. MD2 := MBUF[sid];  /*Temp. 2 gets destination offset address.
MD1 := MD1 + MD2;  /*Calculate address of destination operand.
MRB := MD1;  /*Request destination operand.
CT := CT - 1, MD3 := MD1, MD1 := MD3;
/*Switch registers.  MD1 gets source operand.  MD3 gets address.
e. MD2 := MBUF[sid];  /*Temp. 2 gets destination operand.
MD1 := MD1 <op> MD2;  /*Perform operation.
CC := <op> CC;  /*Update flags.
MRB := PC, PC := PC + 2;  /*Request instruction.
MWB := (MD3, MD1);  /*Write results.
CT := CT - 1.  /*Decrement counter.
```

Process Control (PCNTL) Class
RESET instruction format



Instruction = RESET

RTL description

all modes

a. IR := MBUF[sid]; /*Load instruction.

IF PRIORITY := 0 THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;

ELSE

{ SET RESETLINE; /*RESETLINE will be set.

When RESETLINE is set, it starts a counter

which will reset RESETLINE after eight clocks. The

counter loads the present count (0-7) into each stream's

Stream Identification number (SID), which initializes all

eight SID's over the eight clock cycles. In addition, all

ERR[sid] bits will be cleared, and the full bits for MBUF,

the Memory Read Buffer, the Memory Write Buffer, will all

be cleared, which will cancel all memory requests. Each process will

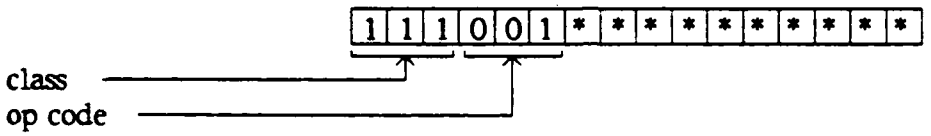
then execute NOP's until it arrives in Segment One. In Segment One,

a START instruction is loaded into the stream's Instruction Register.

The START instruction is described in the STRAP class of instructions.

}

Process Control (PCNTL) Class
SPL instruction format



Instruction = SPL

RTL description

all modes

a. IR := MBUF[sid]; /*Load instruction.

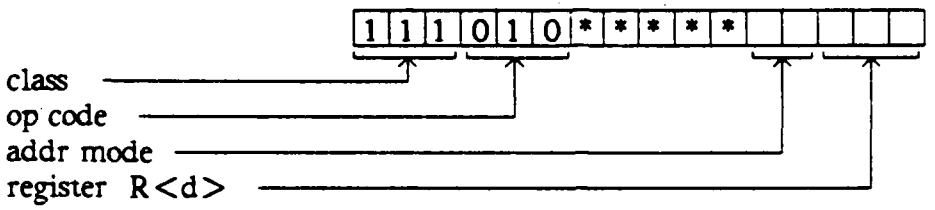
CT := 1; /*Set counter.

PRIORITY := 0; /*Set priority lower.

MRB := PC. PC := PC + 2; /*Request next instruction.

CT := CT - 1; /*Decrement counter.

Process Control (PCNTL) Class
HLTP instruction format



Instruction = HLTP

RTL description

mode 0

a. IR := MBUF[sid]; /*Load instruction.

IF PRIORITY = 0 THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;

ELSE

{ MD2 := R<d>, CT := 3; /*Temp. 1 has Process I.D.

CT := CT - 1. /*Decrement counter.

b. IF STOP = 1 OR PID = HLTB

THEN CT := 1. GO TO HALTED PROCESS INSTRUCTION ROUTINE;

ELSE

{ HLTB := MD2, HT := 1, HBFULL := 1;

/*Halt buffer loaded and set HT flag.

CT := CT - 1. /*Decrement counter.

c. V := HT, HT := 0, HBFULL := 0, HLTB := 0;

/*Overflow set if process not found.

MRB := PC, PC := PC - 2; /*Request next instruction.

CT := CT - 1. /*Decrement counter.

}

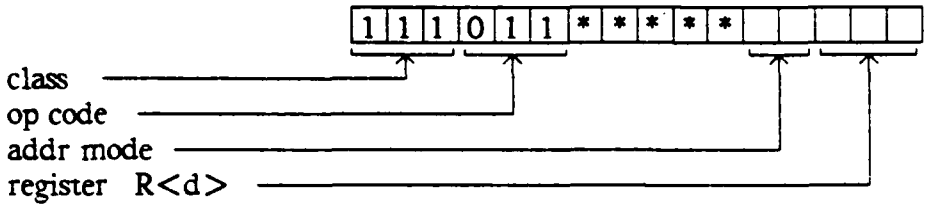
}

```
mode 1

        a. IR := MBUF[sid]; /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
  {     MD1 := R<d>, CT := 4;  /*Temp. 1 has address of Process LD.
        MRB := MD1;  /*Request Process LD.
        CT := CT - 1.  /*Decrement counter.
        b. MD2 := MBUF[sid]; /*Temp. 2 gets PID of process to be halted.
        CT := CT - 1;  /*Decrement counter.
        c. IF STOP = 1 OR PID = HLTB
                THEN CT := 1.  GO TO HALTED PROCESS INSTRUCTION ROUTINE;
                ELSE
                {  HLTB := MD2, HT := 1, HBFULL := 1;
                   /*Load Process LD. into halt buffer.  Set buffer flags.
                   CT := CT - 1;  /*Decrement counter.
                d. V := HT, HT := 0, HBFULL := 0, HLTB := 0;
                   /*Overflow set if process not found.
                   MRB := PC, PC := PC + 2;  /*Request instruction.
                   CT := CT - 1.  /*Decrement counter.
                }
  }

mode 2

        a. IR := MBUF[sid]; /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1,  GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
  {     MD1 := R<d>, CT := 4;  /*Temp. 1 has address of Process LD.
        MRB := MD1;  /*Request Process LD.
        CT := CT - 1.  /*Decrement counter.
        b. MD2 := MBUF[sid]; /*Temp. 2 gets PID of process to be halted.
        MD1 := MD1 + 2;  /*Increment register value.
```

```
        R<d> := MD1;  /*Update register.
        CT := CT - 1;  /*Decrement counter.
      c. IF STOP = 1 OR PID = HLTB
            THEN CT := 1,  GO TO HALTED PROCESS INSTRUCTION ROUTINE;
            ELSE
            {  HLTB := MD2, HT := 1, HBFULL := 1;
               /*Load Process I.D. into halt buffer.  Set buffer flags.
               CT := CT - 1;  /*Decrement counter.
            d. V := HT, HT := 0, HBFULL := 0, HLTB := 0;
               /*Overflow set if process not found.
               MRB := PC, PC := PC + 2;  /*Request instruction.
               CT := CT - 1.  /*Decrement counter.
          }
   }

mode 3
      a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
  {     MD1 := R<d>, CT := 5;  /*Temp. 1 gets base address.
        MRB := PC, PC := PC + 2;  /*Request offset.
        CT := CT - 1.  /*Decrement counter.
      b. MD2 := MBUF[sid];  /*Temp. 2 gets offset.
        MD1 := MD1 + MD2;  /*Compute address of Process I.D.
        MRB := MD1;  /*Request Process I.D.
        CT := CT - 1;  /*Decrement counter.
      c. MD2 := MBUF[sid];  /*Temp. 2 gets PID of process to be halted.
        CT := CT - 1;  /*Decrement counter.
      d. IF STOP = 1 OR PID = HLTB
          THEN CT := 1,  GO TO HALTED PROCESS INSTRUCTION ROUTINE;
          ELSE
```

```
{  HLTB := MD2, HT := 1, HBFULL := 1;
      /*Load Process LD. into halt buffer.
      CT := CT - 1;  /*Decrement counter.
   e. V := HT, HT := 0, HBFULL :=0, HLTB := 0;
      /*Overflow set if process not found.
      MRB := PC, PC := PC + 2;  /*Request instruction.
      CT := CT - 1.  /*Decrement counter.
}
}
```

Process Control (PCNTL) Class
CPSW instruction format



Instruction = CPSW

RTL description

mode 0

a. IR := MBUF[sid]; /*Load instruction.

IF PRIORITY = 0 THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;

ELSE

{ MD2 := R<d>, CT := 1; /*Temp. 2 gets new process status word.

PS := MD2; /*PS reg. gets new process status word.

MRB := PC, PC := PC + 2; /*Request instruction.

CT := CT - 1. /*Decrement counter.

}

mode 1

a. IR := MBUF[sid]; /*Load instruction.

IF PRIORITY = 0 THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;

ELSE

{ MD1 := R<d>, CT := 2; /*Temp. 1 has address of process status word.

MRB := MD1; /*Request new process status word.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Temp. 2 has new process status word.

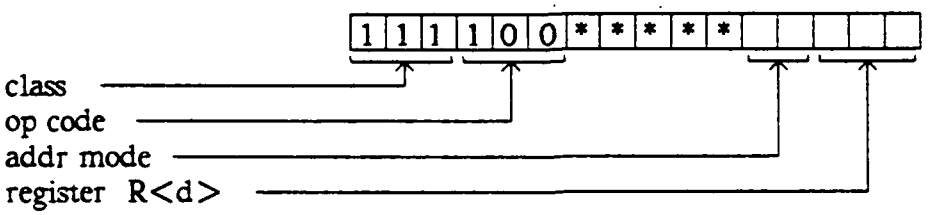
PS := MD2; /*PS reg. gets new process status word.

MRB :=PC, PC := PC + 2; /*Request next instruction.

```
            CT := CT - 1.  /*Decrement counter.
    }
mode 2
        a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1. GO TO ILLEGAL INSTRUCTION ROUTINE;
    ELSE
    {       MD1 := R<d>, CT := 2; /*Temp. 1 has address of process status word.
            MRB := MD1, MD1 := MD1 + 2;  /*Request new process status word.
            R<d> := MD1;  /*Autoincrement register value.
            CT := CT - 1;  /*Decrement counter.
        b. MD2 := MBUF[sid];  /*Temp. 2 has new process status word.
            PS := MD2;  /*PS reg. gets new process status word.
            MRB :=PC, PC := PC + 2;  /*Request next instruction.
            CT := CT - 1.  /*Decrement counter.
mode 3
        a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1. GO TO ILLEGAL INSTRUCTION ROUTINE;
    ELSE
    {       MD1 := R<d>, CT := 3;  /*Temp. 1 is base address.
            MRB := PC, PC := PC + 2;  /*Request offset.
            CT := CT - 1;  /*Decrement counter.
        b. MD2 := MBUF[sid];  /*Temp. 2 has offset address.
            MD1 := MD1 - MD2;  /*Compute address of new process status word.
            MRB := MD1;  /*Request new process status word.
            CT := CT - 1;  /*Decrement counter.
        c. MD2 := MBUF[sid];  /*Temp. 2 gets new process status word.
            PS := MD2;  /*PS reg. gets new process status word.
            MRB := PC, PC := PC + 2;  /*Request next instruction.
            CT := CT - 1.  /*Decrement counter.
```

Process Control (PCNTL) Class

RPSW instruction format



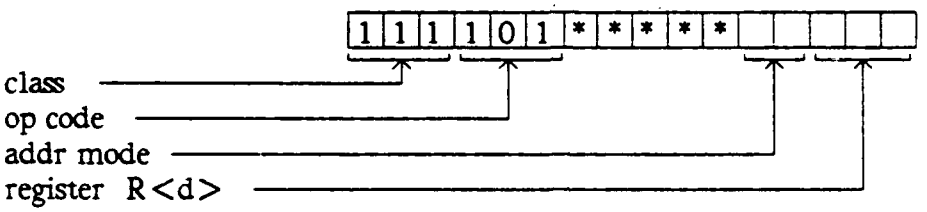
Instruction = RPSW

RTL description

```
mode 0   a. IR := MBUF[sid]; /*Load instruction.
             MD2 := PS, CT := 1; /*Temp. 2 gets value of PS reg.
             MRB := PC, PC := PC + 2; /*Request next instruction.
             CT := CT - 1. /*Decrement counter.

mode 1   a. IR := MBUF[sid]; /*Load instruction.
             MD1 := R<d>, CT := 1; /*Temp. 1 gets destination address.
             MD2 := PS; /*Load PSW value into Temp. 2.
             MRB := PC, PC := PC + 2; /*Request next instruction.
             MWB := (MD1, MD2); /*Write PSW into memory destination.
             CT := CT - 1; /*Decrement counter.

mode 2   a. IR := MBUF[sid]; /*Load instruction.
             MD1 := R<d>, CT := 2; /*Temp. 1 gets destination address.
             MD2 := PS; /*Load PSW value into Temp. 2.
             MWB := (MD1, MD2); /*Write PSW into memory destination.
             CT := CT - 1; /*Decrement counter.
            b. MD1 := MD1 + 2; /*Autoincrement register value.
             MRB := PC, PC := PC + 2; /*Request next instruction.
             R<d> := MD1; /*Autoincrement register.
```

```
                CT := CT - 1;  /*Decrement counter.

mode 3      a. IR := MBUF[sid];  /*Load instruction.
                MD1 := R<d>, CT := 3;  /*Temp. 1 is base address.
                MRB := PC, PC := PC + 2;  /*Request offset.
                CT := CT - 1;  /*Decrement counter.
            b. MD2 := MBUF[sid];  /*Temp. 2 has offset address.
                MD1 := MD1 + MD2;  /*Compute address of destination.
                CT := CT - 1;  /*Decrement counter.
            c. MD2 := PS;  /*Temp. 2 gets value of PSW.
                MRB := PC, PC := PC + 2;  /*Request next instruction.
                MWB := (MD1, MD2);  /*Write PSW value to destination.
                CT := CT - 1.  /*Decrement counter.
```

Process Control (PCNTL) Class
TSET instruction format



Instruction = TSET

RTL description

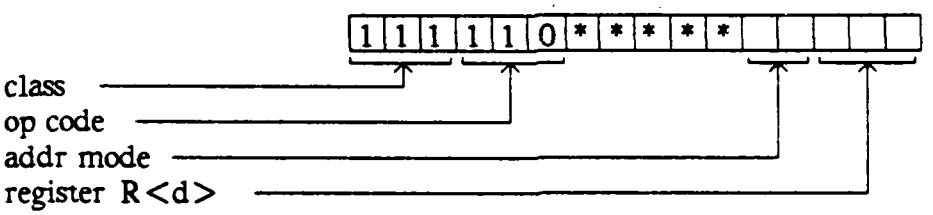
```
mode 0    a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE:
  ELSE
  {       MD1 := R<d>, CT := 2;  /*Temp. 1 gets semaphore destination.
          MRB := MD1;  /*Request semaphore value and set it if not set.
          CT := CT - 1;  /*Decrement counter.
        b. MD2 := MBUF[sid];  /*Read semaphore value.
          Z := TEST MD2;  /*Test to see if semaphore was clear.
          IF Z = 0  THEN
                { MRB := MD1;  /*Read semaphore again.
                  NOP Segment 6-8;  GO TO CYCLE b.
                  /*Repeats cycle b. until semaphore register is cleared.
                }
          ELSE  { MRB := PC; PC := PC + 2;  /*Semaphore register was clear.
                  CT := CT - 1; /*Semaphore register was clear, so continue.
                }
  }

mode 1    a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE:
```

```
ELSE
{       MD1 := R<d>, CT := 3; /*MD1 gets address of semaphore destination.
        MRB := MD1;  /*Request semaphore destination.
        CT := CT - 1;  /*Decrement counter.
      b. MD2 := MBUF[sid];  /*Read semaphore destination.
        MRB := MD2;  /*Request semaphore value and set it if not set.
        CT := CT - 1, MD1 := MD2;  /*Temp. 1 gets destination.
      c. MD2 := MBUF[sid];  /*Read semaphore value.
        Z := TEST MD2;  /*Test to see if semaphore was clear.
        IF Z = 0  THEN
                {  MRB := MD1;   /*Read semaphore again.
                   NOP Segment 6-8;  GO TO CYCLE b.
                   /*Repeats cycle b. until semaphore register is cleared.
                }
        ELSE   {  MRB := PC; PC := PC + 2;  /*Semaphore register was clear.
                   CT := CT - 1; /*Semaphore register was clear, so continue.
                }
}

mode 2   a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
{       MD1 := R<d>, CT := 3; /*MD1 gets address of semaphore destination.
        MRB := MD1, MD1 := MD1 + 2;  /*Request semaphore destination.
        R<d> := MD1;  /*Autoincrement register value.
        CT := CT - 1;  /*Decrement counter.
      b. MD2 := MBUF[sid];  /*Read semaphore destination.
        MRB := MD2;  /*Request semaphore value and set it if not set.
        CT := CT - 1, MD1 := MD2;  /*Temp. 1 gets destination.
      c. MD2 := MBUF[sid];  /*Read semaphore value.
        Z := TEST MD2;  /*Test to see if semaphore was clear.
```

```
        IF Z = 0 THEN
                { MRB := MD1;  /*Read semaphore again.
                  NOP Segment 6-8; GO TO CYCLE b.
                  /*Repeats cycle b. until semaphore register is cleared.
                }
        ELSE  { MRB := PC; PC := PC + 2;  /*Semaphore register was clear.
                  CT := CT - 1; /*Semaphore register was clear, so continue.
                }
}

mode 3   a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
{       MD1 := R<d>, CT := 4; /*MD1 gets base address of semaphore.
        MRB := PC, PC := PC + 2;  /*Request offset.
        CT := CT - 1.  /*Decrement counter.
       b. MD2 := MBUF[sid];  /*Temp. 2 gets offset.
        MD1 := MD1 + MD2;  /*Compute address of semaphore destination.
        MRB := MD1;  /*Request semaphore destination.
        CT := CT - 1;  /*Decrement counter.
       c. MD2 := MBUF[sid];  /*Read semaphore destination.
        MRB := MD2;  /*Request semaphore value and set it if not set.
        CT := CT - 1, MD1 := MD2;  /*Temp. 1 gets destination.
       d. MD2 := MBUF[sid];  /*Read semaphore value.
        Z := TEST MD2;  /*Test to see if semaphore was clear.
        IF Z = 0  THEN
                { MRB := MD1;  /*Read semaphore again.
                  NOP Segment 6-8;  GO TO CYCLE b.
                  /*Repeats cycle b. until semaphore register is cleared.
                }
```

```
ELSE   { MRB := PC; PC := PC + 2; /*Semaphore register was clear.
         CT := CT - 1; /*Semaphore register was clear, so continue.
       }
}
```

Process Control (PCNTL) Class
CTST instruction format



Instruction = CTST

RTL description

mode 0 a. IR := MBUF[sid]; /*Load instruction.

IF PRIORITY = 0 THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;

ELSE

{ MD1 := R<d>, CT := 2; /*Temp. 1 gets semaphore destination.

CT := CT - 1, MD3 := MD1; /*Temp. 3 gets semaphore destination.

b. MD1 := <CLEAR> MD1; /*Set register to zero for clear.

MRB := PC, PC := PC - 2; /*Request next instruction.

MWB := (MD3,MD1); /*Clear semaphore register.

CT := CT - 1; /*Decrement counter.

}

mode 1 a. IR := MBUF[sid]; /*Load instruction.

IF PRIORITY = 0 THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;

ELSE

{ MD1 := R<d>, CT := 3; /*MD1 gets address of semaphore destination.

MRB := MD1; /*Request semaphore destination.

CT := CT - 1; /*Decrement counter.

b. MD2 := MBUF[sid]; /*Read semaphore destination.

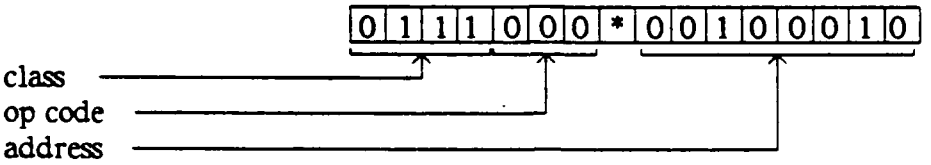
MD1 := <CLEAR> MD1; /*Set register to zero for clear.

CT := CT - 1, MD3 := MD2; /*Temp. 3 gets destination.

```
        c. MRB := PC, PC := PC + 2;  /*Request next instruction.
           MWB := (MD3, MD1);  /*Clear semaphore register.
           CT := CT - 1;  /*Decrement counter.
  }

mode 2   a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
  {        MD1 := R<d>, CT := 3; /*MD1 gets address of semaphore destination.
           MRB := MD1, MD1 := MD1 + 2;  /*Request semaphore destination.
           R<d> := MD1;  /*Autoincrement register value.
           CT := CT - 1;  /*Decrement counter.
         b. MD2 := MBUF[sid];  /*Read semaphore destination.
           MD1 := <CLEAR> MD1;  /*Set register to zero for clear.
           CT := CT - 1, MD3 := MD2;  /*Temp. 1 gets destination.
         c. MRB := PC, PC := PC + 2;  /*Request next instruction.
           MWB := (MD3, MD1);  /*Clear semaphore register.
           CT := CT - 1;  /*Decrement counter.
  }

mode 3   a. IR := MBUF[sid];  /*Load instruction.
IF PRIORITY = 0  THEN IIT := 1, GO TO ILLEGAL INSTRUCTION ROUTINE;
  ELSE
  {        MD1 := R<d>, CT := 3; /*MD1 gets base address of semaphore.
           MRB := PC, PC := PC + 2;  /*Request offset.
           CT := CT - 1.  /*Decrement counter.
         b. MD2 := MBUF[sid];  /*Temp. 2 gets offset.
           MD1 := MD1 + MD2;  /*Compute address of semaphore destination.
           MRB := MD1;  /*Request semaphore destination.
           CT := CT - 1;  /*Decrement counter.
         c. MD2 := MBUF[sid];  /*Read semaphore destination.
```

```
    MD1 := <CLEAR > MD1;  /*Set register to zero for clear.
    CT := CT - 1, MD3 := MD2;  /*Temp. 3 gets destination.
d.  MRB := PC, PC := PC + 2;  /*Request next instruction.
    MWB := (MD3, MD1);  /*Clear semaphore register.
    CT := CT - 1;  /*Decrement counter.
}
```

System Trap (STRAP) Class
Interrupt Instruction (INTR) format



RTL description

In Segment 5, the following conditions are checked.

The code below will be executed when the conditions are met.

IF CT = 1 AND PRIORITY = 0 AND (INTERRUPT = MEMERR = STOP = IIT

= TRACE = HBF = WRW = DW = 0) AND INTLINE SET

THEN

In seg. 5 INTERRUPT = 1, INTLINE := 0; /*Sets flag in status word, clears interrupt line.

The MRB := PC, PC = PC - 2 instruction is suppressed.

Seg. 6-8 Code is executed as usual depending on the instruction.

Starting on the next cycle, this code is executed when CT = 0 AND

INTERRUPT = 1 AND ERR[sid] = 0.

a. IR := [Interrupt Instruction]; /*Load interrupt opcode.

CT := 4; /*Set counter.

MRB := PC; /*Dummy read--check for memory errors.

CT := CT - 1; /*Decrement counter.

b. IF ERR[sid] = 1

THEN GO TO MEMORY ERROR ROUTINE;

ELSE CONTINUE

MD1 := R6; /*Temp. 1 is SP.

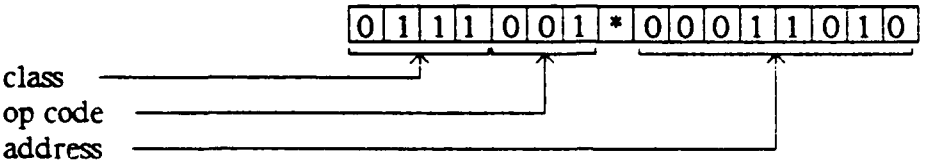
MD1 := MD1 - 2; /*Decrement SP.

MRB := IR[7:0], IR := IR - 2; /*Request PS from trap vector.

MWB := (MD1, PS); /*Push PS.

```
    R6 := MD1;  /*Update SP.
    CT := CT - 1;  /*Decrement counter.
c. MD2 := MBUF[sid];  /*Temp. 2 gets new PS.
    MD1 := MD1 - 2;  /*Decrement SP.
    PS := MD2, PRIORITY := 1;  /*Load in new PS.
    MRB := IR[7:0];  /*Request new PC.
    MWB := (MD1, PC);  /*Push old PC.
    R6 := MD1;  /* Update SP.
    CT := CT - 1;  /*Decrement counter.
d. MD2 := MBUF[sid];  /*Temp. 2 gets new PC.
    PC := MD2;  /*Load in new PC.
    MRB := PC, PC := PC + 2, INTERRUPT := 0;  /*Request next instruction.
    CT := CT - 1.  /*Decrement counter.
```

System Trap (STRAP) Class
Halted Process Instruction format



RTL description

In Segment 1, HLTB is checked against each stream's PID.

The code below will be executed when the conditions are met.

IF HLTB = PID AND (INTERRUPT = MEMERR = IIT = 0)

THEN

In seg. 1 STOP := 1, HT := 0; /*Sets flag in status word.

/*Clears buffer flag to indicate process was found.

The rest of the code for the current instruction is executed as

normal, except for seg. 5 in the last cycle of the instruction.

In seg. 5 IF CT = 1 AND STOP = 1 AND MEMERR = IIT = 0

THEN The MRB := PC, PC = PC - 2 instruction is suppressed.

Seg. 6-8 Code is executed as usual depending on the instruction.

Starting on the next cycle, this code is executed when CT = 0 AND

STOP=1 AND MEMERR = IIT = ERR[sid] = 0.

a. IR := [Halted Process Instruction], STOP := 0; /*Load opcode.

CT := 4; /*Set counter.

MRB := PC; /*Dummy read--check for memory errors.

CT := CT - 1; /*Decrement counter.

b. IF ERR[sid] = 1

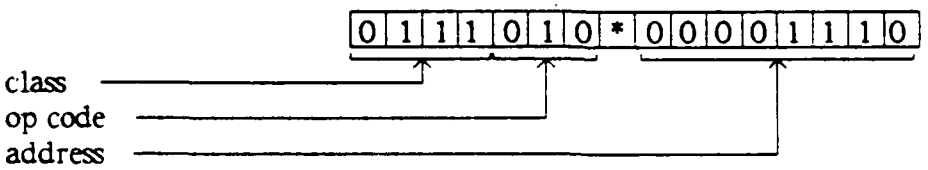
THEN GO TO MEMORY ERROR ROUTINE;

ELSE CONTINUE

MD1 := R6; /*Temp. 1 is SP.

```
MD1 := MD1 - 2; /*Decrement SP.
MRB := IR[7:0], IR := IR - 2; /*Request PS from trap vector.
MWB := (MD1, PS); /*Push PS.
R6 := MD1; /*Update SP.
CT := CT - 1; /*Decrement counter.
c. MD2 := MBUF[sid]; /*Temp. 2 gets new PS.
MD1 := MD1 - 2; /*Decrement SP.
PS := MD2, PRIORITY := 1; /*Load in new PS.
MRB := IR[7:0]; /*Request new PC.
MWB := (MD1, PC); /*Push old PC.
R6 := MD1; /* Update SP.
CT := CT - 1; /*Decrement counter.
d. MD2 := MBUF[sid]; /*Temp. 2 gets new PC.
PC := MD2; /*Load in new PC.
MRB := PC, PC := PC + 2; /*Request next instruction.
CT := CT - 1. /*Decrement counter.
```

System Trap (STRAP) Class
Trace Trap Instruction format



RTL description

In Segment 5, the following conditions are checked.

The code below will be executed when the conditions are met.

IF CT = 1 AND T = 1 AND STAT = 0 AND NOT (RTT or TSET instruction) AND

(INTLINE = 0 OR PRIORITY = 1)

THEN

In seg. 5 TRACE = 1 ; /*Sets flag in status word.

The MRB := PC. PC = PC + 2 instruction is suppressed.

Seg. 6-8 Code is executed as usual depending on the instruction.

Starting on the next cycle, this code is executed when CT = 0 AND

TRACE = 1 AND ERR[sid] = 0.

a. IR := [Trace Trap (instruction]; /*Load trace trap opcode.

CT := 4; /*Set counter.

MRB := PC; /*Dummy read--check for memory errors.

CT := CT - 1; /*Decrement counter.

b. IF ERR[sid] = 1

THEN GO TO MEMORY ERROR ROUTINE;

ELSE CONTINUE

MD1 := R6; /*Temp. 1 is SP.

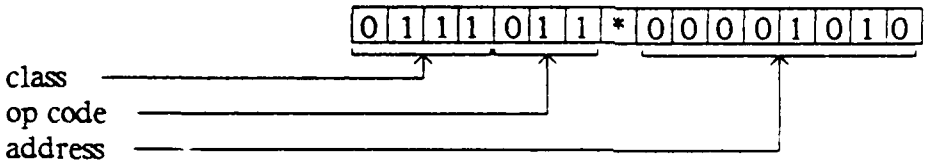
MD1 := MD1 - 2; /*Decrement SP.

MRB := IR[7:0], IR := IR - 2; /*Request PS from trap vector.

MWB := (MD1, PS); /*Push PS.

```
R6 := MD1;  /*Update SP.

CT := CT - 1;  /*Decrement counter.

c. MD2 := MBUF[sid];  /*Temp. 2 gets new PS.

MD1 := MD1 - 2;  /*Decrement SP.

PS := MD2, PRIORITY := 1;  /*Load in new PS.

MRB := IR[7:0];  /*Request new PC.

MWB := (MD1, PC);  /*Push old PC.

R6 := MD1;  /* Update SP.

CT := CT - 1;  /*Decrement counter.

d. MD2 := MBUF[sid];  /*Temp. 2 gets new PC.

PC := MD2;  /*Load in new PC.

MRB := PC, PC := PC + 2, TRACE = 0;  /*Request next instruction.

CT := CT - 1.  /*Decrement counter.
```

System Trap (STRAP) Class
Illegal Instruction Trap format



RTL description

In Segment 2, the instructions are checked to see if they are legal.

If the opcodes are not legal, or if the priority is not valid for a particular

instruction, or if an addressing mode is not legal, IIT is set to 1.

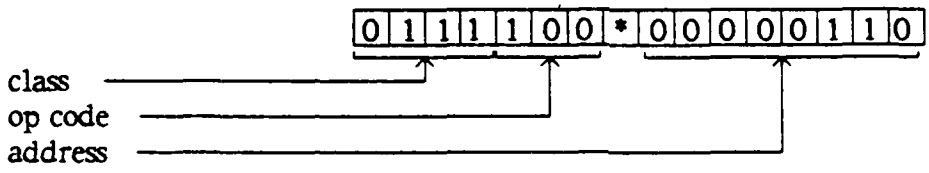
The rest of that cycle is NOP'd, and at the beginning of the next cycle,

since IIT is 1, this code is executed.

```
IF IIT = 1  AND ERR[sid] = 0
  THEN
      a. IR := [Illegal Instruction Trap], STAT = 0;
        /*Load trap opcode, clear flag since code is started.
        CT := 4;  /*Set counter.
        MRB := PC;  /*Dummy read--check for memory errors.
        CT := CT - 1;  /*Decrement counter.
      b. IF ERR[sid] = 1
        THEN GO TO MEMORY ERROR ROUTINE;
        ELSE CONTINUE
        MD1 := R6;  /*Temp. 1 is SP.
        MD1 := MD1 - 2;  /*Decrement SP.
        MRB := IR[7:0], IR := IR - 2;  /*Request PS from trap vector.
        MWB := (MD1, PS);  /*Push PS.
        R6 := MD1;  /*Update SP.
        CT := CT - 1;  /*Decrement counter.
```

```
c. MD2 := MBUF[sid]; /*Temp. 2 gets new PS.
   MD1 := MD1 - 2; /*Decrement SP.
   PS := MD2, PRIORITY := 1; /*Load in new PS.
   MRB := IR[7:0]; /*Request new PC.
   MWB := (MD1, PC); /*Push old PC.
   R6 := MD1; /* Update SP.
   CT := CT - 1; /*Decrement counter.
d. MD2 := MBUF[sid]; /*Temp. 2 gets new PC.
   PC := MD2; /*Load in new PC.
   MRB := PC, PC := PC + 2; /*Request next instruction.
   CT := CT - 1. /*Decrement counter.
```

System Trap (STRAP) Class
Memory Error Trap format



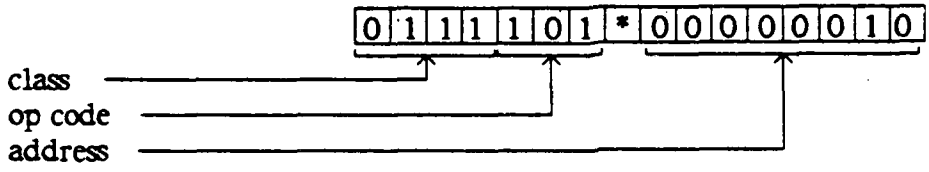
RTL description

In Segment 1, if data is expected from memory, MBUF[sid] is checked to see whether a memory error occurred. If a memory error occurred, ERR[sid] (which is a bit associated with that buffer location) will be 1.

```
IF ERR[sid] = 1
  THEN
      a. IR := [Memory Error Trap], MEMERR := 1, ERR[sid] := 0,
        STOP := IIT := TRACE := HBF := WRW := RRW := DW := 0;
        /*Load trap opcode and clear out other status bits.
        IF INTERRUPT = 1   THEN
        {    INTLINE := 1, INTERRUPT := 0, MD1 := R6, CT := 3; }
        /*Set interrupt line again since process will not finish interrupt.
        ELSE  MD1 := R6, CT := 3; /*Temp. 1 gets stack pointer, regardless.
        MD1 := MD1 - 2;  /*Decrement SP.
        MRB := IR[7:0], IR := IR - 2;  /*Request PS from trap vector.
        MWB := (MD1, PS);  /*Push PS.
        R6 := MD1;  /*Update SP.
        CT := CT - 1;  /*Decrement counter.
      b. MD2 := MBUF[sid];  /*Temp. 2 gets new PS.
        MD1 := MD1 - 2;  /*Decrement SP.
        PS := MD2, PRIORITY := 1;  /*Load in new PS.
        MRB := IR[7:0];  /*Request new PC.
```

```
MWB := (MD1, PC); /*Push old PC.
R6 := MD1; /* Update SP.
CT := CT - 1; /*Decrement counter.
c. MD2 := MBUF[sid]; /*Temp. 2 gets new PC.
PC := MD2; /*Load in new PC.
MRB := PC, PC := PC + 2, MEMERR := 0; /*Request next instruction.
CT := CT - 1. /*Decrement counter.
```

System Trap (STRAP) Class
START Trap instruction format



RTL description

If a RESET instruction occurs and sets RESETLINE or the RESETLINE pin is set from off-chip, the CT registers for all processes will be set to 0.

All ERR[sid] bits will be cleared out.

The processes will then execute NOP's until the process gets to the first segment. When the process gets to the first segment, a START instruction is loaded into its instruction register. A counter is started which will reset RESETLINE after eight clocks. This counter also loads the Stream Identification number (SID) for each of the streams. This instruction overrides all other traps or interrupts.

```
IF RESETLINE = 1   THEN
        a. IR := [START Trap], STAT := 0;  /*Load opcode.
           CT := 3;  Set counter.
           PRIORITY := 1;  /*Set priority bit.
           MRB := IR[7:0], IR := IR - 2;  /*Request PS from trap vector.
           CT := CT - 1;  /*Decrement counter.
        b. MD2 := MBUF[sid];  /*Temp. 2 gets new PS.
           PS := MD2, PRIORITY := 1;  /*Load in new PS.
           MRB := IR[7:0];  /*Request PC from trap vector.
           CT := CT - 1;  /*Decrement counter.
        c. MD2 := MBUF[sid];  /*Temp. 2 gets new PC.
           PC := MD2;  /*Load in new PC.
```

```
MRB := PC. PC := PC + 2;  /*Request next instruction.

CT := CT - 1.  /*Decrement counter.
```

APPENDIX D. CONDITIONS FOR THE MICROCONTROL

This is a table which lists each microinstruction and the conditions under which it is executed. This list includes the conditions that would be inputs to a PLA decoder. It is not a reduced table, and there are cases where not all of the conditions listed would have to be included as inputs to the decoder. These conditions are listed for completeness, and the actual implementation is left to the designer. This list does not explicitly list the other hardware actions that occur independently of the PLA. This includes such things as the automatic setting of the status bits HBF, WRW, RRW, and DW, depending on whether the corresponding buffer is full, and the action of the reset counter, which is automatically set when the RESETLINE is initially set. These independent hardware actions are adequately described by the main text of the thesis.

The main input conditions which are included in all segments are the Instruction Register (IR), the Cycle Counter (CT), the Dynamic Status Word (STAT), and the Priority bit (P). The Instruction Register includes the class, instruction, and the source and destination register modes. There are other conditions which are included for only some of the segments. These include the PID match bit, the ERR[sid] bit, INTLINE, and the Trace (T) bit, and the Zero (Z) bit from the Process Status Word (PS). Although RESETLINE is not specifically listed as an input in the table, it is an input to all of the PLA's. RESETLINE is 0 throughout the table except where it is specifically stated that RESETLINE is set. Near the bottom of each segment description is a section that describes the action when RESETLINE is set. In the table, if there is a blank, that field does not exist for that particular instruction. A "-" means that it is a "don't care" situation. Only the instructions where some action occurs are explicitly listed in the table. It is assumed that if none of the conditions listed apply to a given segment, that segment is NOP'd. Only special cases of NOP's are explicitly listed.

D.1 Operations for Segment 1

In Segment One, the inputs to the decoder are basically the same as the other segments. The Instruction Register, which includes the class, opcode, source and destination addressing modes (whenever they exist), as well as the Cycle Counter (CT), and the Dynamic Status Word (STAT), are all used to determine which microoperation is to be executed. In Segment One, however, MBUF must be checked to see whether there is an error flag (ERR[sid]) set for a stream. In addition to this, HLTB must be checked to see whether there is a match with a

process's PID. If there is a matched PID, the normal microinstruction will be executed; however, in addition, some flags will be set in parallel with the normal instruction. This instruction is included near the end of the microinstructions for Segment 1, but it does occur in parallel with the normal instructions. If the PID matches, a one is present in the table. If the error flag is set for a stream, ERR[sid] will be one in the table below.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
-	-	-	-	0	00000000-	-	-	0

Operation is: IR := MBUF[sid]

When conditions are:

STRAP	INTR			0	100000000	-	-	0

Operation is: IR := [Interrupt Instruction]

When conditions are:

STRAP	HPRO			0	001000000	-	-	0

Operation is: IR := [Halted Process Instruction], STOP := 0

When conditions are:

STRAP	TRTR			0	000010000	-	-	0

Operation is: IR := [Trace Trap Instruction]

When conditions are:

STRAP	ILIT			-	00-1——	-	-	0

Operation is: IR := [Illegal Instruction Trap], STAT := 0

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
STRAP	MERR			-	-0——	-	-	1

Operation is: IR := [Memory Error Trap], MEMERR := 1, ERR[sid] := 0
STOP := IIT := TRACE := HBF := WRW := RRW := DW := 0

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
SOP	all		1	1	00-00000-	-	-	0
SOP	all		2	2	00-00000-	-	-	0
SOP	all		3	2	00-00000-	-	-	0
SOP	all		3	1	00-00000-	-	-	0
UTRAP	all			2	00-00000-	-	-	0
UTRAP	all			1	00-00000-	-	-	0
RETI	all			2	00-00000-	1	-	0
RETI	all			1	00-00000-	1	-	0
JUMP	JMP		3	2	00-00000-	-	-	0
JUMP	RTS			2	00-00000-	-	-	0
JUMP	JSR		3	2	00-00000-	-	-	0
DOP	all	0	1	1	00-00000-	-	-	0
DOP	all	0	2	2	00-00000-	-	-	0
DOP	all	0	3	2	00-00000-	-	-	0
DOP	all	0	3	1	00-00000-	-	-	0
DOP	all	1	0	1	00-00000-	-	-	0
DOP	all	1	1	2	00-00000-	-	-	0
DOP	all	1	1	1	00-00000-	-	-	0
DOP	all	1	2	3	00-00000-	-	-	0
DOP	all	1	2	2	00-00000-	-	-	0
DOP	all	1	3	3	00-00000-	-	-	0
DOP	all	1	3	2	00-00000-	-	-	0
DOP	all	1	3	1	00-00000-	-	-	0
DOP	all	2	0	1	00-00000-	-	-	0
DOP	all	2	1	2	00-00000-	-	-	0
DOP	all	2	1	1	00-00000-	-	-	0
DOP	all	2	2	3	00-00000-	-	-	0
DOP	all	2	2	2	00-00000-	-	-	0

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
DOP	all	2	3	3	00-00000-	-	-	0
DOP	all	2	3	2	00-00000-	-	-	0
DOP	all	2	3	1	00-00000-	-	-	0
DOP	all	3	0	2	00-00000-	-	-	0
DOP	all	3	0	1	00-00000-	-	-	0
DOP	all	3	1	3	00-00000-	-	-	0
DOP	all	3	1	2	00-00000-	-	-	0
DOP	all	3	1	1	00-00000-	-	-	0
DOP	all	3	2	4	00-00000-	-	-	0
DOP	all	3	2	3	00-00000-	-	-	0
DOP	all	3	2	2	00-00000-	-	-	0
DOP	all	3	3	4	00-00000-	-	-	0
DOP	all	3	3	3	00-00000-	-	-	0
DOP	all	3	3	2	00-00000-	-	-	0
DOP	all	3	3	1	00-00000-	-	-	0
PCNTL	HLTP		1	3	00-00000-	1	-	0
PCNTL	HLTP		2	3	00-00000-	1	-	0
PCNTL	HLTP		3	4	00-00000-	1	-	0
PCNTL	HLTP		3	3	00-00000-	1	-	0
PCNTL	CPSW		1	1	00-00000-	1	-	0
PCNTL	CPSW		2	1	00-00000-	1	-	0
PCNTL	CPSW		3	2	00-00000-	1	-	0
PCNTL	CPSW		3	1	00-00000-	1	-	0
PCNTL	RPSW		3	2	00-00000-	-	-	0
PCNTL	TSET		-	1	00-00000-	1	-	0
PCNTL	TSET		1	2	00-00000-	1	-	0
PCNTL	TSET		2	2	00-00000-	1	-	0
PCNTL	TSET		3	3	00-00000-	1	-	0
PCNTL	TSET		3	2	00-00000-	1	-	0
PCNTL	CTST		1	2	00-00000-	1	-	0
PCNTL	CTST		2	2	00-00000-	1	-	0
PCNTL	CTST		3	3	00-00000-	1	-	0
PCNTL	CTST		3	2	00-00000-	1	-	0

Operation is: MD2 := MBUF[sid]

In one of the HLTP microinstructions, the Halt buffer (HLTB) is loaded with another process's PID. It is loaded only if the process executing the HLTP is not about to be halted itself. The STOP bit must be 0 and the PID must not match the PID in the Halt Buffer (if any) in order to execute the microinstruction that loads another PID into the Halt Buffer. If STOP = 1, the Cycle Counter is set to 1, and it executes the microinstructions for that cycle. If the PID matches, the Cycle Counter is set to 1 and the flags for the STOP are set. If neither of these conditions is true, then the Halt Buffer is checked to see whether it is full. If it is full, the HBF flag in the process's STAT word is set, and the Halt Buffer is not loaded. The process passes through the pipe until the Halt Buffer is empty. This happens in hardware when the microinstruction is executed, so is not explicitly mentioned below.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
PCNTL	HLTP		-	2	00000-00-	1	0	0

Operation is: HLTB := MD2, HT := 1, HBFULL := 1

When conditions are:

PCNTL	HLTB		-	1	00000000-	1	0	0

Operation is: V := HT, HT := 0, HBFULL := 0, HLTB := 0

The following operation can occur in parallel with any of the other instructions already mentioned.

When conditions are:

-	-	-	-	-	00-0——	-	1	0

Operation is: STOP := 1, HT := 0

In the HLTP instruction, the process cannot stop another process when it is being stopped itself. To avoid this, the Cycle Counter is set to one. This effectively causes a jump to the last cycle in the instruction, where the process is stopped. This occurs in parallel with setting the flags to indicate that the process is being stopped, if it just identified its PID.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
PCNTL	HLTP		-	2	00100-00-	1	0	0
PCNTL	HLTP		-	2	00000-00-	1	1	0

Operation is: CT := 1

When RESETLINE line is set, the operations below are executed instead of the operations that were shown above.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
-	-	-	-	-	-	-	-	-

Operation is: IR := [START Trap], STAT := 0

If none of the conditions above are true, or if the conditions listed directly below are true, this segment is NOP'd.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	PID Match	ERR [sid]
-	-	-	-	-	—0-0100	-	0	0
-	-	-	-	-	—0-0010	-	0	0

Operation is: NOP

D.2 Operations for Segment 2

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
SOP	all		all	0	00-000000	-
JUMP	JMP		1	0	00-000000	-
JUMP	JMP		2	0	00-000000	-
JUMP	JMP		3	0	00-000000	-
JUMP	JSR		1	1	00-000000	-
JUMP	JSR		2	2	00-000000	-
JUMP	JSR		3	2	00-000000	-
DOP	all	0	1	0	00-000000	-
DOP	all	0	2	0	00-000000	-
DOP	all	0	3	0	00-000000	-
DOP	all	1	0	1	00-000000	-
DOP	all	1	1	3	00-000000	-
DOP	all	1	2	4	00-000000	-
DOP	all	1	3	0	00-000000	-
DOP	all	2	0	1	00-000000	-
DOP	all	2	1	0	00-000000	-
DOP	all	2	2	4	00-000000	-
DOP	all	2	3	0	00-000000	-
DOP	all	3	0	1	00-000000	-
DOP	all	3	1	3	00-000000	-
DOP	all	3	2	4	00-000000	-
DOP	all	3	3	3	00-000000	-
PCNTL	HLTP		1	0	00-000000	1
PCNTL	HLTP		2	0	00-000000	1
PCNTL	HLTP		3	0	00-000000	1
PCNTL	CPSW		1	0	00-000000	1
PCNTL	CPSW		2	0	00-000000	1
PCNTL	CPSW		3	0	00-000000	1
PCNTL	RPSW		1	0	00-000000	-
PCNTL	RPSW		2	0	00-000000	-
PCNTL	RPSW		3	0	00-000000	-
PCNTL	TSET		-	0	00-000000	1
PCNTL	CTST		-	0	00-000000	1

Operation is: MD1 := R < d >

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
DOP	all	0	1	0	00-000000	-
DOP	all	0	2	0	00-000000	-
DOP	all	0	3	1	00-000000	-
DOP	all	1	0	0	00-000000	-
DOP	all	1	1	0	00-000000	-
DOP	all	1	2	0	00-000000	-
DOP	all	1	3	2	00-000000	-
DOP	all	2	0	0	00-000000	-
DOP	all	2	1	2	00-000000	-
DOP	all	2	2	0	00-000000	-
DOP	all	2	3	2	00-000000	-
DOP	all	3	0	0	00-000000	-
DOP	all	3	1	0	00-000000	-
DOP	all	3	2	0	00-000000	-
DOP	all	3	3	0	00-000000	-

Operation is: MD1 := R<s>

When conditions are:

UTRAP	all			0	00-000000	-
RETI	all			0	00-000000	1
STRAP	INTR			3	000000000	-
STRAP	HPRO			3	000000000	-
STRAP	TRTR			3	00-010000	-
STRAP	ILIT			3	000000000	-
STRAP	MERR			0	010000000	-

Operation is: MD1 := R6

When conditions are:

LOOP	SOB			0	00-000000	-
PCNTL	HLTP		0	0	00-000000	1
PCNTL	CPSW		0	0	00-000000	1

Operation is: MD2 := R<d>

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT.	Prt.
DOP	all	0	0	0	00-000000	-

Operation is: MD2 := R<s>, MD1 := R<d>

When conditions are:

JUMP	RTS		0	00-000000	-
JUMP	JSR	1	0	00-000000	-
JUMP	JSR	2	0	00-000000	-
JUMP	JSR	3	0	00-000000	-

Operation is: MD2 := R<s>, MD1 := R6

When conditions are:

STRAP	MERR	0	110000000	-

Operation is: INTLINE := 1, INTERRUPT := 0, MD1 := R6

When conditions are:

RETI	all		0	00-000000	0
JUMP	JMP	0	-	00-000000	-
JUMP	JSR	0	-	00-000000	-
PCNTL	RESET		0	00-000000	0
PCNTL	HLTP	-	0	00-000000	0
PCNTL	CPSW	-	0	00-000000	0
PCNTL	TSET	-	0	00-000000	0
PCNTL	CTST	-	0	00-000000	0

Operation is: IIT := 1

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
PCNTL	RESET			0	00-000000	1

Operation is: SET RESETLINE

The Cycle Counter can be set in parallel with any of the instructions above. The instructions to set the Cycle Counter are found below.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
SOP	all		0	0	00-000000	-
BRAN	all			0	00-000000	-
COND	all			0	00-000000	-
JUMP	JMP		1	0	00-000000	-
DOP	all	0	0	0	00-000000	-
PCNTL	SPL			0	00-000000	-
PCNTL	CPSW		0	0	00-000000	1
PCNTL	RPSW		0	0	00-000000	-
PCNTL	RPSW		1	0	00-000000	-

Operation is: CT := 1

When conditions are:

SOP	all		1	0	00-000000	-
LOOP	SOB			0	00-000000	-
JUMP	JMP		2	0	00-000000	-
JUMP	RTS			0	00-000000	-
JUMP	JSR		1	0	00-000000	-
DOP	all	0	1	0	00-000000	-
DOP	all	1	0	0	00-000000	-
DOP	all	2	0	0	00-000000	-
PCNTL	CPSW		1	0	00-000000	1
PCNTL	CPSW		2	0	00-000000	1
PCNTL	RPSW		2	0	00-000000	-
PCNTL	TSET		0	0	00-000000	1
PCNTL	CTST		0	0	00-000000	1

Operation is: CT := 2

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Prt.
SOP	all		2	0	00-000000	-
SOP	all		3	0	00-000000	-
UTRAP	all			0	00-000000	-
RETI	all			0	00-000000	1
JUMP	JMP		3	0	00-000000	-
JUMP	JSR		2	0	00-000000	-
JUMP	JSR		3	0	00-000000	-
DOP	all	0	2	0	00-000000	-
DOP	all	0	3	0	00-000000	-
DOP	all	1	1	0	00-000000	-
DOP	all	2	1	0	00-000000	-
DOP	all	3	0	0	00-000000	-
PCNTL	HLTP		0	0	00-000000	1
PCNTL	CPSW		3	0	00-000000	1
PCNTL	RPSW		3	0	00-000000	-
PCNTL	TSET		1	0	00-000000	1
PCNTL	TSET		2	0	00-000000	1
PCNTL	CTST		1	0	00-000000	1
PCNTL	CTST		2	0	00-000000	1
STRAP	MERR			0	-10000000	-
STRAP	START			0	000000000	-

Operation is: CT := 3

When conditions are:

DOP	all	1	2	0	00-000000	-
DOP	all	1	3	0	00-000000	-
DOP	all	2	2	0	00-000000	-
DOP	all	2	3	0	00-000000	-
DOP	all	3	1	0	00-000000	-
PCNTL	HLTP		1	0	00-000000	1
PCNTL	HLTP		2	0	00-000000	1
PCTNL	TSET		3	0	00-000000	1
PCTNL	CTST		3	0	00-000000	1
STRAP	INTR			0	100000000	-

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
STRAP	HPRO			0	000000000	-
STRAP	TRTR			0	00-010000	-
STRAP	ILIT			0	000000000	-

Operation is: CT := 4

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
DOP	all	3	2	0	00-000000	-
DOP	all	3	3	0	00-000000	-
PCNTL	HLTP		3	0	00-000000	1

Operation is: CT := 5

When RESETLINE line is set, the only instruction that can be executed is the START instruction. All other instructions are NOP'd. The instruction execution when the RESETLINE is set is listed below.

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
STRAP	START			0	000000000	-

Operation is: CT := 3

When RESETLINE is set and conditions are:

NOT START	-	-	-	———	-	

Operation is: NOP

If none of the conditions above are true, or if the conditions listed directly below are true, this segment is NOP'd.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	—0-0001	-
-	-	-	-	-	—0-0010	-
-	-	-	-	-	—0-0100	-
PCNTL	HLTP		-	-	0—0-1000	1

Operation is: NOP

D.3 Operations for Segment 3

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
SOP	all		0	1	00-000000	-

Operation is: MD1 := <op> MD1

When conditions are:

PCNTL	CTST		0	1	00-000000	1
PCNTL	CTST		1	2	00-000000	1
PCNTL	CTST		2	2	00-000000	1
PCNTL	CTST		3	2	00-000000	1

Operation is: MD1 := <CLEAR> MD1

When conditions are:

SOP	all		1	1	00-000000	-
SOP	all		2	2	00-000000	-
SOP	all		3	1	00-000000	-

Operation is: MD1 := <op> MD2

When conditions are:

DOP	all	0	1	1	00-000000	-
DOP	all	0	2	2	00-000000	-
DOP	all	0	3	1	00-000000	-
DOP	all	1	1	1	00-000000	-
DOP	all	1	2	2	00-000000	-
DOP	all	2	2	2	00-000000	-
DOP	all	3	1	1	00-000000	-
DOP	all	3	2	2	00-000000	-
DOP	all	3	3	1	00-000000	-

Operation is: MD1 := MD1 <op> MD2

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
DOP	all	0	0	1	00-000000	-
DOP	all	1	0	1	00-000000	-
DOP	all	1	3	1	00-000000	-
DOP	all	2	0	1	00-000000	-
DOP	all	2	1	1	00-000000	-
DOP	all	2	3	1	00-000000	-
DOP	all	3	0	1	00-000000	-

Operation is: MD1 := MD2 <op> MD1

When conditions are:

SOP	all		3	2	00-000000	-
JUMP	JMP		3	2	00-000000	-
JUMP	JSR		3	2	00-000000	-
DOP	all	0	3	2	00-000000	-
DOP	all	1	3	3	00-000000	-
DOP	all	2	3	3	00-000000	-
DOP	all	3	0	2	00-000000	-
DOP	all	3	1	3	00-000000	-
DOP	all	3	2	4	00-000000	-
DOP	all	3	3	4	00-000000	-
DOP	all	3	3	2	00-000000	-
PCNTL	HLTP		4	3	00-000000	1
PCNTL	CPSW		3	2	00-000000	1
PCNTL	RPSW		3	2	00-000000	-
PCNTL	TSET		3	3	00-000000	1
PCNTL	CTST		3	3	00-000000	1

Operation is: MD1 := MD1 + MD2

When conditions are:

SOP	all	2	1	00-000000	-
JUMP	JMP	2	1	00-000000	-
JUMP	JSR	2	1	00-000000	-

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Prt.
DOP	all	0	2	1	00-000000	-
DOP	all	1	2	1	00-000000	-
DOP	all	2	2	1	00-000000	-
DOP	all	3	2	1	00-000000	-
PCNTL	HLTP		2	3	00-000000	-
PCNTL	RPSW		2	1	00-000000	-

Operation is : MD1 := MD1 - 2

When conditions are:

UTRAP	all			3	00-000000	-
UTRAP	all			2	00-000000	-
JUMP	JSR		1	2	00-000000	-
JUMP	JSR		2	3	00-000000	-
JUMP	JSR		3	3	00-000000	-
STRAP	INTR			3	100000000	-
STRAP	INTR			2	100000000	-
STRAP	HPRO			3	000000000	-
STRAP	HPRO			2	000000000	-
STRAP	TRTR			3	00-010000	-
STRAP	TRTR			2	00-010000	-
STRAP	ILIT			3	000000000	-
STRAP	ILIT			2	000000000	-
STRAP	MERR			3	010000000	-
STRAP	MERR			2	010000000	-

Operation is: MD1 := MD1 - 2

When conditions are:

LOOP	SOB			2	00-000000	-

Operation is: MD2 := MD2 - 1

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
JUMP	JMP		1	1	00-000000	-
JUMP	JMP		2	2	00-000000	-
JUMP	JMP		3	1	00-000000	-
JUMP	JSR		1	1	00-000000	-
JUMP	JSR		2	2	00-000000	-
JUMP	JSR		3	1	00-000000	-

Operation is: PC := MD1

When conditions are:

UTRAP	all			1	00-000000	-
RETI	all			2	00-000000	1
JUMP	RTS			2	00-000000	-
STRAP	INTR			1	100000000	1
STRAP	HPRO			1	000000000	1
STRAP	TRTR			1	00-010000	1
STRAP	ILIT			1	000000000	1
STRAP	MERR			1	010000000	1
STRAP	START			1	000000000	1

Operation is: PC := MD2

When conditions are:

PCNTL	RPSW	0	1	00-000000	-
PCNTL	RPSW	1	1	00-000000	-
PCNTL	RPSW	2	2	00-000000	-
PCNTL	RPSW	3	1	00-000000	-

Operation is: MD2 := PS

When conditions are:

BRAN	all	1	00-000000	-

Operation is: IF condition is true THEN PC := PC + 2* IR[7:0]

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
LOOP	SOB			1	00-000000	-

Operation is: IF Z = 0 THEN PC := PC - 2* IR[8:3]

When RESETLINE line is set in the third segment, all instructions are NOP'd.

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	———	-

Operation is: NOP

If none of the conditions above are true, or if the conditions listed directly below are true, this segment is NOP'd.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	—1----	-
-	-	-	-	-	——0001	-
-	-	-	-	-	——0010	-
-	-	-	-	-	-——0100	-
PCNTL	HLTP		-	-	0——1000	-

Operation is: NOP

D.4 Operations for Segment 4

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
SOP	all		0	1	00-000000	-
SOP	all		1	1	00-000000	-
SOP	all		2	2	00-000000	-
SOP	all		3	1	00-000000	-
DOP	all	0	0	1	00-000000	-
DOP	all	0	1	1	00-000000	-
DOP	all	0	2	2	00-000000	-
DOP	all	0	3	1	00-000000	-
DOP	all	1	0	1	00-000000	-
DOP	all	1	1	1	00-000000	-
DOP	all	1	2	2	00-000000	-
DOP	all	1	3	1	00-000000	-
DOP	all	2	0	1	00-000000	-
DOP	all	2	1	1	00-000000	-
DOP	all	2	2	2	00-000000	-
DOP	all	2	3	1	00-000000	-
DOP	all	3	0	1	00-000000	-
DOP	all	3	1	1	00-000000	-
DOP	all	3	2	2	00-000000	-
DOP	all	3	3	1	00-000000	-

Operation is: CC := CC <op> CC

When conditions are:

COND	all			1	00-000000	-

Operation is: PS[code] := IR[4]

When conditions are:

RETI	all			1	00-000000	1
PCNTL	CPSW		-	1	00-000000	1

Operation is: PS := MD2

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
UTRAP	all			2	00-000000	-
STRAP	INTR			2	100000000	-
STRAP	HPRO			2	000000000	-
STRAP	TRTR			2	00-010000	-
STRAP	ILIT			2	000000000	-
STRAP	MERR			2	010000000	-
STRAP	START			2	000000000	1

Operation is: PS := MD2, PRIORITY := 1

When conditions are:

PCNTL	SPL			1	00-000000	-

Operation is: PRIORITY := 0

When conditions are:

STRAP	START			3	000000000	-

Operation is: PRIORITY := 1

When conditions are:

LOOP	SOB			2	00-000000	-
PCNTL	TSET		-	1	00-000000	1

Operation is: Z := TEST MD2

When RESETLINE line is set, the only instruction that can be executed is the START instruction. All other instructions are NOP'd. The instruction execution when the RESETLINE is set is listed below.

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
STRAP	START			3	000000000	-

Operation is: PRIORITY := 1

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
NOT START	-	-	-	——	-	

Operation is: NOP

If none of the conditions above are true, or if the conditions listed directly below are true, this segment is NOP'd.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	—1——	-
-	-	-	-	-	——0001	-
-	-	-	-	-	——0010	-
-	-	-	-	-	——0100	-
PCNTL	HLTP				0—1000	1

Operation is: NOP

D.5 Operations for Segment 5

In Segment Five, several conditions are checked in the last cycle of an instruction before the operations occur. This is done so that interrupts, traces, and halted processes can be handled properly. Besides the regular Instruction Register, Cycle Counter, and Priority bit, the Interrupt line (INTLINE abbreviated as INTL in the table below) and the trace bit (T) are also used to determine the instruction. In two instructions, the Zero bit (Z) is also needed to determine the instruction. It is included in the table for the microinstructions in Segment Five.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	Z
-	NOT TSET	-	-	1	0000000-0	-	0	0	-
-	NOT TSET	-	-	1	0000000-0	1	1	0	-
PCNTL	TSET		-	1	0000000-0	1	-	0	1
RETI	RTT			1	0000000-0	1	-	-	-
SOP	all		3	3	00-0000-0	-	-	-	-
JUMP	JMP		3	3	00-0000-0	-	-	-	-
JUMP	JSR		3	3	00-0000-0	-	-	-	-
DOP	all	0	3	3	00-0000-0	-	-	-	-
DOP	all	1	3	4	00-0000-0	-	-	-	-
DOP	all	2	3	4	00-0000-0	-	-	-	-
DOP	all	3	0	3	00-0000-0	-	-	-	-
DOP	all	3	1	4	00-0000-0	-	-	-	-
DOP	all	3	2	5	00-0000-0	-	-	-	-
DOP	all	3	3	5	00-0000-0	-	-	-	-
DOP	all	3	3	3	00-0000-0	-	-	-	-
PCNTL	HLTP		3	5	00-0000-0	1	-	-	-
PCNTL	CPSW		3	3	00-0000-0	1	-	-	-
PCNTL	RPSW		3	3	00-0000-0	-	-	-	-
PCNTL	TSET		3	4	00-0000-0	1	-	-	-
PCNTL	CTST		3	4	00-0000-0	1	-	-	-

Operation is: MRB := PC, PC := PC + 2

There are some instructions that include one microinstruction in addition to the MRB := PC, PC := PC + 2 (instruction fetch) listed above when the Cycle Counter is 1. These instructions are in the System Trap class, and

the additional microinstruction involves modifying one of the status bits. These instructions are listed below, showing the additional microinstruction in parallel with the normal instruction fetch. Some of the conditions may be a subset of the conditions listed above for the instruction fetch.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	Z
STRAP	INTR			1	1000000-0	1	-	-	-

Operation is: MRB := PC, PC := PC + 2, INTERRUPT := 0

When conditions are:

STRAP	TRTR			1	00-0100-0	1	-	-	-

Operation is: MRB := PC, PC := PC + 2, TRACE := 0

When conditions are:

STRAP	MERR			1	0100000-0	1	-	-	-

Operation is: MRB := PC, PC := PC + 2, MEMERR := 0

When conditions are:

STRAP	INTR			4	1000000-0	-	-	-	-
STRAP	HPRO			4	0000000-0	-	-	-	-
STRAP	TRTR			4	00-0100-0	-	-	-	-
STRAP	ILIT			4	0000000-0	-	-	-	-

Operation is: MRB := PC

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	Z
SOP	all		1	2	00-0000-0	-	-	-	-
SOP	all		2	3	00-0000-0	-	-	-	-
SOP	all		3	2	00-0000-0	-	-	-	-
DOP	all	0	1	2	00-0000-0	-	-	-	-
DOP	all	0	2	3	00-0000-0	-	-	-	-
DOP	all	0	3	2	00-0000-0	-	-	-	-
DOP	all	1	0	2	00-0000-0	-	-	-	-
DOP	all	1	1	3	00-0000-0	-	-	-	-
DOP	all	1	1	2	00-0000-0	-	-	-	-
DOP	all	1	2	4	00-0000-0	-	-	-	-
DOP	all	1	2	3	00-0000-0	-	-	-	-
DOP	all	1	3	3	00-0000-0	-	-	-	-
DOP	all	1	3	2	00-0000-0	-	-	-	-
DOP	all	2	1	3	00-0000-0	-	-	-	-
DOP	all	2	2	3	00-0000-0	-	-	-	-
DOP	all	2	3	3	00-0000-0	-	-	-	-
DOP	all	3	0	2	00-0000-0	-	-	-	-
DOP	all	3	1	3	00-0000-0	-	-	-	-
DOP	all	3	1	2	00-0000-0	-	-	-	-
DOP	all	3	2	4	00-0000-0	-	-	-	-
DOP	all	3	2	3	00-0000-0	-	-	-	-
DOP	all	3	3	4	00-0000-0	-	-	-	-
DOP	all	3	3	2	00-0000-0	-	-	-	-
PCNTL	HLTP		1	4	00-0000-0	1	-	-	-
PCNTL	HLTP		2	4	00-0000-0	1	-	-	-
PCNTL	HLTP		3	4	00-0000-0	1	-	-	-
PCNTL	CPSW		1	2	00-0000-0	1	-	-	-
PCNTL	CPSW		3	2	00-0000-0	1	-	-	-
PCNTL	TSET		0	2	00-0000-0	1	-	-	-
PCNTL	TSET		1	3	00-0000-0	1	-	-	-
PCNTL	TSET		3	3	00-0000-0	1	-	-	-
PCNTL	TSET		-	1	0000000-0	1	-	-	0
PCNTL	CTST		1	3	00-0000-0	1	-	-	-
PCNTL	CTST		3	3	00-0000-0	1	-	-	-

Operation is: MRB := MD1

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	Z
RETI	all			3	00-0000-0	1	-	-	-
RETI	all			2	00-0000-0	1	-	-	-
JUMP	all			2	00-0000-0	-	-	-	-
DOP	all	2	0	2	00-0000-0	-	-	-	-
DOP	all	2	1	2	00-0000-0	-	-	-	-
DOP	all	2	2	4	00-0000-0	-	-	-	-
DOP	all	2	3	2	00-0000-0	-	-	-	-
PCNTL	CPSW		2	2	00-0000-0	1	-	-	-
PCNTL	TSET		2	3	00-0000-0	1	-	-	-
PCNTL	CTST		2	3	00-0000-0	1	-	-	-

Operation is: MRB := MD1. MD1 := MD1 - 2

When conditions are:

PCNTL	TSET		1	2	00-0000-0	1	-		-
PCNTL	TSET		2	2	00-0000-0	1	-		-
PCNTL	TSET		3	2	00-0000-0	1	-		-

Operation is: MRB := MD2

When conditions are:

UTRAP	all			2	00-0000-0	-	-	-	-
STRAP	INTR			2	0000000-0	1	-	-	-
STRAP	HPRO			2	0000000-0	1	-	-	-
STRAP	TRTR			2	00-0100-0	1	-	-	-
STRAP	ILIT			2	0000000-0	1	-	-	-
STRAP	MERR			2	0100000-0	1	-	-	-
STRAP	START			2	0000000-0	1	-	-	-

Operation is: MRB := IR[7:0]

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	Z
UTRAP	all			3	00-0000-0	-	-	-	-
STRAP	INTR			3	1000000-0	-	-	-	-
STRAP	HPRO			3	0000000-0	-	-	-	-
STRAP	TRTR			3	00-0100-0	-	-	-	-
STRAP	ILIT			3	0000000-0	-	-	-	-
STRAP	MERR			3	0100000-0	-	-	-	-
STRAP	START			3	0000000-0	-	-	-	-

Operation is: MRB := IR[7:0], IR := IR - 2

When conditions are:

-	-	-	-	1	0000000-0	0	1	-	-

Operation is: INTERRUPT = 1, INTLINE := 0

When conditions are:

-	not(RTT OR TSET)	-	-	1	000000000	-	0	1	-
-	not(RTT OR TSET)	-	-	1	000000000	1	-	1	-

Operation is: TRACE := 1

When RESETLINE line is set, the only instruction that can be executed is the START instruction. All other instructions are NOP'd. The instruction execution when the RESETLINE is set is listed below.

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	Z
STRAP	START			0	000000000	1	-	-	-

Operation is: MRB := IR[7:0], IR := IR - 2

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	Z
-	NOT START	-	-	-	——	-	-	-	-

Operation is: NOP

If none of the conditions above are true, or if the conditions listed directly below are true, this segment is NOP'd.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	INTL	T	
-	-	-	-	-	——0001	-	-	-	-
-	-	-	-	-	——0100	-	-	-	-
PCNTL	HLTP		-	-	0——1000	1	-	-	-
-	-	-	-	1	0010——	-	-	-	-
-	-	-	-	-	-0-1——	-	-	-	-

Operation is: NOP

D.6 Operations for Segment 6

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
SOP	all		1	1	-0-0-0-00	-
SOP	all		2	2	-0-0-0-00	-
SOP	all		3	1	-0-0-0-00	-
DOP	all	0	1	1	-0-0-0-00	-
DOP	all	0	2	2	-0-0-0-00	-
DOP	all	0	3	1	-0-0-0-00	-
DOP	all	1	1	1	-0-0-0-00	-
DOP	all	1	2	2	-0-0-0-00	-
DOP	all	1	3	1	-0-0-0-00	-
DOP	all	2	1	1	-0-0-0-00	-
DOP	all	2	2	2	-0-0-0-00	-
DOP	all	2	3	1	-0-0-0-00	-
DOP	all	3	1	1	-0-0-0-00	-
DOP	all	3	2	2	-0-0-0-00	-
DOP	all	3	3	1	-0-0-0-00	-
PCNTL	CTST		-	1	-0-0-0-00	-

Operation is: MWB := (MD3,MD1)

When conditions are:

JUMP	JSR		1	2	-0-0-0-00	-
JUMP	JSR		2	3	-0-0-0-00	-
JUMP	JSR		3	3	-0-0-0-00	-
PCNTL	RPSW		1	1	-0-0-0-00	-
PCNTL	RPSW		2	1	-0-0-0-00	-
PCNTL	RPSW		3	1	-0-0-0-00	-

Operation is: MWB := (MD1,MD2)

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
UTRAP	all			3	-0-0-0-00	-
STRAP	INTR			3	100000-00	-
STRAP	HPRO			3	000000-00	-
STRAP	TRTR			3	00-010-00	-
STRAP	ILIT			3	000000-00	-
STRAP	MERR			3	010000-00	-

Operation is: MWB := (MD1, PS)

When conditions are:

UTRAP	all			2	-0-0-0-00	-
STRAP	INTR			2	100000-00	-
STRAP	HPRO			2	000000-00	-
STRAP	TRTR			2	00-010-00	-
STRAP	ILIT			2	000000-00	-
STRAP	MERR			2	010000-00	-

Operation is: MWB := (MD1, PC)

When RESETLINE line is set in the sixth segment, all instruction are NOP'd.

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	------	

Operation is: NOP

If none of the conditions above are true, or if the conditions listed below are true, this segment is NOP'd.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	——0001	-
-	-	-	-	-	——0010	-
PCNTL	HLTP		-	-	0——1000	1
-	-	-	-	-	—1——	-
PCNTL	TSET		-	1	00-00——	-

Operation is: NOP

D.7 Operations for Segment 7

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
SOP	all		0	1	-0-0-0000	-
SOP	all		2	1	-0-0-0000	-
JUMP	JMP		2	1	-0-0-0000	-
JUMP	JSR		2	1	-0-0-0000	-
DOP	all	0	0	1	-0-0-0000	-
DOP	all	0	2	1	-0-0-0000	-
DOP	all	1	2	1	-0-0-0000	-
DOP	all	2	2	1	-0-0-0000	-
DOP	all	3	0	1	-0-0-0000	-
DOP	all	3	2	1	-0-0-0000	-
PCNTL	HLTP		2	3	00-0-0000	1
PCNTL	CPSW		2	2	00-0-0000	1
PCNTL	RPSW		2	1	-0-0-0000	-
PCNTL	TSET		2	3	00-0-0000	1
PCNTL	CTST		2	3	00-0-0000	1

Operation is: R<d> := MD1

When conditions are:

LOOP	SOB			2	-0-0-0000	-
DOP	all	1	0	1	-0-0-0000	-
DOP	all	2	0	1	-0-0-0000	-

Operation is: R<d> := MD2

When conditions are:

DOP	all	2	0	2	-0-0-0000	-
DOP	all	2	1	2	-0-0-0000	-
DOP	all	2	2	4	-0-0-0000	-
DOP	all	2	3	2	-0-0-0000	-

Operation is: R<s> := MD1

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
JUMP	RTS			1	-0-0-0000	-
JUMP	JSR		1	1	-0-0-0000	-
JUMP	JSR		2	2	-0-0-0000	-
JUMP	JSR		3	1	-0-0-0000	-

Operation is: R<s> := MD2

When conditions are:

UTRAP	all			3	-0-0-0000	-
UTRAP	all			2	-0-0-0000	-
RETI	all			3	-0-0-0000	-
RETI	all			2	-0-0-0000	-
JUMP	RTS			2	-0-0-0000	-
JUMP	JSR		1	2	-0-0-0000	-
JUMP	JSR		2	3	-0-0-0000	-
JUMP	JSR		3	3	-0-0-0000	-
STRAP	*INTR*			*3*	*100000000*	-
STRAP	INTR			2	100000000	1
STRAP	HPRO			3	000000000	-
STRAP	HPRO			2	000000000	-
STRAP	TRTR			3	00-010000	-
STRAP	TRTR			2	00-010000	1
STRAP	ILIT			3	000000000	-
STRAP	ILIT			2	000000000	1
STRAP	MERR			3	010000000	-
STRAP	MERR			2	010000000	1

Operation is: R6 := MD1

When RESETLINE line is set in the seventh segment, all instructions are NOP'd.

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	--------	-

Operation is: NOP

If none of the conditions above are true, or if the conditions listed directly below are true, this segment is NOP'd.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.
-	-	-	-	-	——0001	-
-	-	-	-	-	——0010	-
-	-	-	-	-	——0100	-
PCNTL	HLTP		-	-	0——1000	1
-	-	-	-	-	—1——	-

Operation is: NOP

D.8 Operations for Segment 8

In Segment Eight, the Instruction Register, Cycle Counter, and the Priority bit are all checked to determine the microinstruction. In addition to this, for the TSET instruction, it is necessary to check the Zero bit (Z). The Zero bit is included in the table. The Cycle Counter can be decremented in parallel with most of the microinstructions in Segment 8, although it is listed separately near the end.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	Z
SOP	all		1	2	-0-0-0000	-	-
SOP	all		2	3	-0-0-0000	-	-
SOP	all		3	2	-0-0-0000	-	-
DOP	all	0	1	2	-0-0-0000	-	-
DOP	all	0	2	3	-0-0-0000	-	-
DOP	all	0	3	2	-0-0-0000	-	-
DOP	all	1	3	3	-0-0-0000	-	-
DOP	all	2	1	3	-0-0-0000	-	-
DOP	all	2	3	3	-0-0-0000	-	-
PCNTL	CTST		0	2	00-0-0000	1	-

Operation is: MD3 := MD1

When conditions are:

SOP	all	-	2	2	-0-0-0000	-	-
DOP	all	0	2	2	-0-0-0000	-	-
DOP	all	1	2	2	-0-0-0000	-	-
DOP	all	2	2	2	-0-0-0000	-	-
DOP	all	3	2	2	-0-0-0000	-	-

Operation is: MD1 := MD3

When conditions are:

DOP	all	1	3	2	-0-0-0000	-	-
DOP	all	2	3	2	-0-0-0000	-	-
PCNTL	TSET		1	2	00-0-0000	1	-
PCNTL	TSET		2	2	00-0-0000	1	-
PCNTL	TSET		3	2	00-0-0000	1	-

Operation is: MD1 := MD2

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	Z
DOP	all	3	3	3	-0-0-0000	-	-
PCNTL	CTST		1	2	00-0-0000	1	-
PCNTL	CTST		2	2	00-0-0000	1	-
PCNTL	CTST		3	2	00-0-0000	1	-

Operation is: MD3 := MD2

When conditions are:

JUMP	JSR		1	2	-0-0-0000	-	-
JUMP	JSR		2	3	-0-0-0000	-	-
JUMP	JSR		3	2	-0-0-0000	-	-

Operation is: MD2 := PC

When conditions are:

DOP	all	1	1	2	-0-0-0000	-	-
DOP	all	1	2	3	-0-0-0000	-	-
DOP	all	2	2	3	-0-0-0000	-	-
DOP	all	3	1	2	-0-0-0000	-	-
DOP	all	3	2	3	-0-0-0000	-	-

Operation is: MD3 := MD1, MD1 := MD2

When conditions are:

DOP	all	3	3	2	-0-0-0000	-	-

Operation is: MD3 := MD1, MD1 := MD3

The Cycle Counter is decremented in parallel with any of the instructions above. The Cycle Counter is decremented on the last cycle of every instruction, except where the segment is NOP'd because it was an illegal

instruction, or the RESETLINE is set while the process is not executing a START instruction, or the process is waiting for one of the status bits to clear, or the process is waiting for a semaphore to clear. These conditions are listed below.

When conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Prl.	Z
-	-	-	-	-	—1——	-	-
PCNTL	TSET		-	1	00-000000	1	0
-	-	-	-	-	——0001	-	-
-	-	-	-	-	——0010	-	-
-	-	-	-	-	——0100	-	-
PCNTL	HLTP		-	-	0——1000	1	-

Operation is: NOP; the CT := CT - 1 operation does not occur.

If none of the conditions above are true and RESETLINE is not set, then the Cycle Counter will be decremented in parallel with any of the other microinstructions in Segment 8.

If the conditions above are not true:

Operation is: CT := CT - 1

When RESETLINE line is set, the only instruction that can be executed is the START instruction. All other instructions are NOP'd. The instruction execution when the RESETLINE is set is listed below.

When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Prl.	Z
STRAP	START			0	000000000	-	-

Operation is: CT := CT - 1

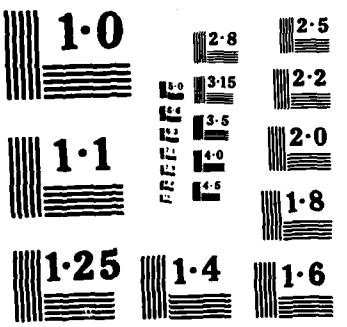
When RESETLINE is set and conditions are:

Instr. Class	Instr.	Src. Mode	Dest. Mode	CT	STAT	Pri.	Z
NOT STRAP	NOT START	-	-	-	——	-	-

Operation is: NOP

AD-A156 314 DESIGN FOR MISP: A MULTIPLE INSTRUCTION STREAM SHARED PIPELINE PROCESSOR(U) ILLINOIS UNIV AT URBANA COMPUTER SYSTEMS GROUP L M PEDERSEN DEC 84 CSG-37 3/3

UNCLASSIFIED N00039-80-C-0556 F/G 9/2 NL



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

REFERENCES

[ARC82]
D. W. Archer, "MISP, A Multiple Instruction Stream Shared Pipeline Microprocessor," Tech. Rept. CSG-9, Coordinated Science Lab., University of Illinois, Urbana, IL, 1982.

[BAN82]
P. Banerjee and J. A. Abraham, "Fault Characterization of VLSI MOS Circuits," *IEEE International Conference on Circuits and Computers*, Sept. 1982, pp. 564-568.

[DAV80]
E. S. Davidson, "A Multiple Stream Microprocessor System: AMP-1," *Seventh Annual Symposium on Computer Architecture*, May 1980, La Baule, France, pp. 9-16.

[DEC81]
Digital Equipment Corporation, *PDP-11 Processor Handbook*, 1981.

[EME78]
J. S. Emer and E. S. Davidson, "Control Store Organization for Multiple Stream Pipelined Processors," *Proc. 1978 Int'l Conf. on Parallel Processing*, Aug. 1978, pp. 43-48.

[EME79]
J. S. Emer, "Shared Resources for Multiple Instruction Stream Pipelined Processors," CSL Report R-838, University of Illinois, Urbana, IL, 1979.

[FIT81]
D. T. Fitzpatrick, et al., "VLSI Implementations of a Reduced Instruction Set Computer," *VLSI Systems and Computations*, H. T. Kung, B. Sproull, and G. Steele, eds., Computer Science Press, Rockville, MD, 1981, pp. 327-336.

[FUC84]
W. K. Fuchs and J. A. Abraham, "A Unified Approach to Concurrent Error Detection in Highly Structured Logic Arrays," *Proc. 14th Annual Int'l Symposium on Fault-Tolerant Computing*, Orlando, FL, 1984, pp. 4-9.

[HEN82]
J. Hennessy, et al., "The MIPS Machine," Digest of papers, *Spring COMPCON 82*, IEEE Computer Society Press, San Francisco, CA, pp. 2-7.

[KAM77]
W. J. Kaminsky, "Architecture for Multiple Instruction Stream LSI Processors," CSL Report R-796, University of Illinois, Urbana, IL, Oct. 1977.

[KAM79]
W. J. Kaminsky and E. S. Davidson, "Developing a Multiple Instruction-Stream Single-Chip Processor," *Computer*, vol. 12, no. 12, Dec. 1979, pp. 66-76.

[MAK83]
G.-P. Mak, "The Design of Programmable Logic Arrays With Concurrent Error Detection," Tech. Rept. CSG-26, Coordinated Science Lab., University of Illinois, Urbana, IL, Dec. 1983.

[PAT82]
J. H. Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Transactions on Computers*, vol. C-31, July 1982, pp. 589-595.

[PTR82]
D. A. Patterson and C. H. Sequin, "A VLSI RISC," *Computer*, vol. 15, no. 9, September 1982, pp. 8-21.

[RAD83]
G. Radin, "The 801 Minicomputer," *IBM Journal of Research and Development*, vol. 27, no. 3, May 1983, pp.237-246.

[SHA74]
L. E. Shar and E. S. Davidson, "A Multiminiprocessor System Implemented Through Pipelining," *Computer*, vol. 7, no. 2, Feb. 1974, pp. 42-51.

[WON83]
C. Y. Wong, W. K. Fuchs, J. A. Abraham, and E. S. Davidson, "The Design of a Microprogram Control Unit with Concurrent Error Detection," *Proc. 13th Annual Int'l Symposium on Fault-Tolerant Computing*, Milano, Italy, June 1983, pp. 476-483.

REPRODUCED AT GOVERNMENT EXPENSE

END

FILMED

8-85

DTIC